University Hannover
Information Systems Institute
Knowledge Based Systems
Prof. Dr. techn. Dipl.-Ing. W. Nejdl
Appelstraße 4
30167 Hannover
Germany

Security and Trust Negotiation in Open Environments

by Sebastian Wittler

ID 2070106

University of Hannover, Germany

seb0815@gmx.de

Bachelor-thesis

05.04.2004-05.08.2004

First Examiner: Prof. Dr. techn. Dipl.-Ing. Wolfgang Nejdl (KBS, L3S)

Second Examiner: Prof. Dr.-Ing. Gabriele von Voigt (RRZN)

Supervisor: Dipl.-Inf. Daniel Olmedilla (KBS, L3S)

Declaration

I declare and assure that I have written this Bachelor-thesis by myself, without any help from other persons.

Further, I declare and assure that I have only used the sources and resources that are listed in the last chapter.

Sebastian Wittler

Table of contents

1. Introduction

Automatic trust negotiation is an emerging field in computer science. Both clients and servers can benefit from it because it can be used to replace the traditional way of registering and login to a server/service. Instead, automatic trust negotiation exchanges rules, policies and credentials between the involved parties in order to minimize effort for the user.

The Learning Lab Lower Saxony (L3S) in Hannover has developed a Java-based prototype named PeerTrust [2,3] that implements trust negotiation supported by a Prolog engine. The tasks in this Bachelor-thesis consisted of increasing security in the current version of the prototype to make it more reliable. Trust negotiation may be used for very sensitive areas like online shops or online banking, so security plays a key role, as hackers shouldn't be able to misuse the system.

The PeerTrust-prototype which contains the solutions developed in this Bachelor-thesis can be found on the included CD, along with source code, documentation and test classes.

2. Trust Negotiation

Compared to the online counterpart, traditional shopping is quite easy. The customer has two possibilities to pay the goods he has chosen. When using cash (coins or banknotes), he gives a certain amount A of it to the salesman and receives back the difference between A and the price sum of all the things he bought, then he is the owner of the purchased goods. Another, more comfortable way is to pay with credit- or EC-card, because it weights less than cash (especially coins). The salesman first checks if the card is valid and has enough credit by examining the hologram and using the scanner, then he prints out a receipt that the customer has to sign. The salesman should compare the signatures on the receipt and on the credit-/EC-card. After passing the security test, the transaction is finished and the customer has paid.

Compared to the shopping described above, using online shops or other client/server-systems that rely on the traditional way of registering and login has some disadvantages [2,3]:

- In registration-forms of many services like online shops, the user has to reveal at least certain information in order to get access to the service. Some of these entries may be of very private nature (for instance the credit card number), but the customer is forced to reveal them if he wants to finish the registration. Often it isn't possible for him to find out if the server can be trusted and if it is secure enough to submit private information. In such traditional client/server-systems, the server should trust the client, not the opposite.
- Even if some entries in registration-forms are required to finish the registration, verifying these information is a very complex (in some cases even impossible) task for the server. Filling out registration forms is often an annoying task and many people abbreviate it by using tools that fill out the entries with false information. What is missing here are digital counterparts of credentials or documents of the real world that prove information about a person (driving license, credit card, ...).
- The user often has to reveal information in the registration phase that seems not to be relevant for the service he wants to use. When the customer wants to visit a website with special articles or information just for registered users, he may find it confusing if he must submit his address and telephone number for a successful registration.
- The user has to keep track of all his login-names and passwords for the online services he uses (assuming he not always uses the same ones for all). Although some of these work with cookies (so the user is automatically logged in when accessing the site, like www.amazon.de), the user may clear the cache (with websites and cookies) of his browser.

2.1 Credentials

When comparing traditional and online shopping, the first one seems to be easier (the annoying registration phase is missing) and more trustworthy. In the real world, special documents (signed by specific authorities) prove that the owner has certain knowledge and skills (certificates, diploma, school report, driving license, ...), personal status (wedding certificate, ...), balance (credit card, ...), identity (identity, ...) etc. Such documents (referred as credentials) are widely accepted and are used to establish trust. If there is a reduced price for students in a cinema, a person must show his university card to be allowed to pay the special price. An applicant who wants to get a job in a company will send them diplomas, certificates and other credentials that prove his (language, working, social, ...) skills. This is the main aspect: because many people are egoistic, look to their own advantage and are therefore dishonest, you can't ask them directly about their skills etc., because they might lie. Instead, you can rely on documents that are signed by specific, independent and trustworthy authorities (schools, universities, etc.) and state something about a person. Coming back to the example with the cinema, everyone can call himself a student, but the special price is only for persons who have a university card. And when the applicant says that he can speak six different languages in order to get a job, the company only believes it when he shows the matching credentials. If the police stands in front of your door and wants to investigate your home, they must show a police badge and a certificate for a police raid. Although some of these credentials may sometimes be faked (this is often hard because of holograms etc.), they are widely accepted and trusted. They help to establish trust between different participants and reduce the risk of cheating.

In traditional client/server-systems (for instance online services or shops), these credentials are missing. Although some services ask for an identity card ID, this is not fully satisfying. What we need are virtual credentials [2,3,7], a digital counterpart of real (paper or card) ones. When a customer wants to access an online service, it can automatically request credentials from the user and get the required information from them (for instance name, address and credit card number of the person), so the user doesn't have to type in the information himself during registration. If there is a special price for students at an online shop, it can request a digital university-credential from the user to find out if he is really one. This all helps to minimize effort for the user. Ideally, he shouldn't do anything when registering; the exchange of the suitable credentials should be done by special negotiation software or agents. Manually registering or having to type in a login and password at online services can be avoided this way; the user doesn't need to keep track of all the passwords and login-names any more.

Digital credentials may contain very sensitive information, so they need to be protected from misuse. Using holograms or other techniques to avoid falsification that work in the real word (for example for credit cards), can't be used here. If a party receives a digital credential, it must be able to find out if the credential really belongs to the party that sent it, if the party that has signed

the credential is really the one specified in the credential and if the content of the credential is valid and has not been tampered with during the transmission. This can be done effectively with the help of digital signatures and asymmetric keys (public and private keys), but this will be described in more detail in the following chapter.

2.2 Rules & Policies

Although this may sound like a great idea, the pure exchange of credentials without any control isn't enough. In the real world, you won't show your credentials to anyone who asks for them, especially when they are sensitive (identity card, credit card, ...). In most cases, you will present them in appropriate situations to people you trust. For digital credentials this shouldn't be different. If the software or agent automatically transmits the credentials to a server without any condition, the user will feel a lack of control over the software and will probably not use it any more. This leads to the conclusion that every party should be able to add rules or policies to the negotiation software in order to protect sensitive resources (for example credentials, documents, data, policies, tokens, services etc.). An online shop for instance may add a policy saying that the shop can only be accessed if the customer presents a valid credit card- and a special member-credential. A customer on the other hand can specify by a policy that his credit card-credential should only be transmitted when the other party has shown a credential that states that it is a member of a trustworthy organization that represents interests of customers. The language used for describing such rules and policies should be powerful, should have well-defined semantics, and at the same time it should not be too complicated, so that even unexperienced persons are able to add their own policies, maybe with the help of special tools that are easy to use but can create complex rules. Issuers of credentials may also give predefined policies that may easily be modified in order to fit the intentions of the user, along with the credential to the user. Logic-based languages like Prolog can be used, because they can be combined with inference engines in the negotiation software and can be extended for the use in trust negotiation. The PeerTrust-prototype for example uses a Prolog-engine and an extended syntax. The expressiveness of languages uses in trust negotiation system differs widely; a common standard is not in sight yet.

2.3 Trust negotiation software

The negotiation software or agents of the participating parties [2,3,7] must understand each other, so they have to be compatible or work on the same protocol. The request for resources, disclosure of policies or credentials etc. is done automatically, but the software may choose between several different ways to achieve its goal. In a trust negotiation, there may be many possibilities of disclosing or requesting policies and credentials to get access to the desired resource, for instance if a client can have more than one way to satisfy a policy that another party has disclosed. The

negotiation software may support different strategies, a cautious one for example expects more requirements before disclosing something as a more offensive strategy. Negotiation software or protocols should not require parties to use only one strategy, as this restricts autonomy, but should also prevent parties from using strategies that don't collaborate. Like the credentials, also the communication channels between the communicating parties should be secure to prevent attackers from listening to the negotiation, manipulating transferred policies and credentials or taking the possession of another party's identity. Own solutions can be implemented to achieve a secure communication, or existing standards like SSL (Secure Sockets Layer) or TLS (Transport Layer Security, a variation of SSL 3.0) may be used for that. SSL and TLS will be described in the next chapter.

2.4 Concept of trust negotiation

All this leads to a possible concept of automatic trust negotiation [2,3,7]. It is not based on the identities of the participating peers, like it is done in many traditional identity-based access controls (a user has to register and login to service, as it is described above). A real negotiation of trust isn't done there (in fact it is only unidirectional), because the client must convince the server to trust him, not the other way round. Instead, the trust negotiation concept described and used in this Bachelor-thesis focuses on the properties of each party, which in turn must be proven to other parties by the disclosure of credentials. A real, bidirectional negotiation is possible; also the client can request the server for showing credentials that support to establish trust between the two parties, which can be done in several steps. As the involved parties are unknown to each other at the beginning, the parties may only want to disclose credentials that are not very sensitive. As the negotiation progresses, each party can request and disclose more and more sensitive credentials, when the other party has satisfied the policies disclosed before. The more time the negotiation takes the more familiar the parties get to each other and the restraint to show sensitive credentials decreases. Another possibility is to involve other parties in the bidirectional communication. Imagine an online shop who offers special prices for students. A customer can only take advantage of them if he presents a credential signed by a university, stating that he is a student there. Maybe the user doesn't possess it, because he lost it or the university doesn't give them to their students. If the online shop wants him to disclose this credential and he doesn't have it, he may ask the university for it, caches it locally perhaps and forwards it back to the shop. So, a third party was involved in the transaction between the online shop and the customer. This feature has also the advantage that every party doesn't need to have stored all his credentials locally, they can be requested from third parties if it is appropriate and if the policies allow this. An example at the end of this chapter will show this. Also, rules of the third party can be cached.

Reconsidering this concept of automatic trust negotiation, it is summarized here again in a

compact form:

- Trust between two parties is not established identity-based, as it is done by traditional client/server-systems, where a client has to manually register and login to a server.
- Instead, the trust negotiation is based on properties which are proven by digital credentials that were issued by specific authorities.
- Digital credentials are the digital counterpart of (paper or card) credentials of the real world, and they should be protected from misuse and tampering.
- Every party can protect their sensitive resources (services, credentials, ...) with custom rules and policies that the other party has to satisfy by the disclosure of special credentials.
- Third parties can be involved during a transaction between two parties, in order to get access to a requested credential or rule that a party doesn't have locally.
- The negotiation (disclosure and exchange of credentials and policies) is performed by a special, smart negotiation software or agent that may support different strategies.
- The negotiation is bidirectional and can consist of more than just one step in order to increase security.
- The negotiation software of parties involved in a negotiation shouldn't be restricted to use the same strategy for disclosing policies and credentials.
- This software minimizes effort for the user by doing the negotiation with the other party automatically, but the user keeps control over the disclosure of his certificates by protecting them with policies.
- The communication between the involved parties should be safe, no attacker should be able to listen or annoy the transaction.

The described concept for automatic trust negotiation is also an access control solution for the Semantic Web, because this is designed to be used not only by humans, it can also accessed by machines or software agents. In contrast to that, the identity-based access controls (traditional registering and login to a server) doesn't fit well here, because it is based on identity and in the Semantic Web many parties interact that do not know each other. The concept of automated trust negotiation, on which this Bachelor-thesis focuses on, is also suitable for peer-to-peer architectures.

2.5 Examples for automatic trust negotiation

The explained concept of automatic trust negotiation will be illustrated by the two following examples, the first one describes a simple negotiation between two parties, the second one is more complex and involves a third party. No special syntax is used for the policies and credentials here, instead, they will be described in words, as this chapter is written in general and should not

use a special language. This is different in the following chapter, where a trust negotiation-prototype is introduced that uses a special language.

2.5.1 First Example

The first example is about a transaction between a customer who wants to buy a ticket for the next game of his favorite soccer club VfB Stuttgart and an online shop that sells tickets for soccer games.

The **customer** has the following credential:

- *VISA-card-credential (signed by VISA, protected by policy1)*

and the following policy:

- *policy1:*
  show *VISA-card-credential only to parties that present a license-credential that is signed by VfB Stuttgart*

The **online shop** has the following credential:

- *license-certificate (signed by VfB Stuttgart, protected by no policy)*

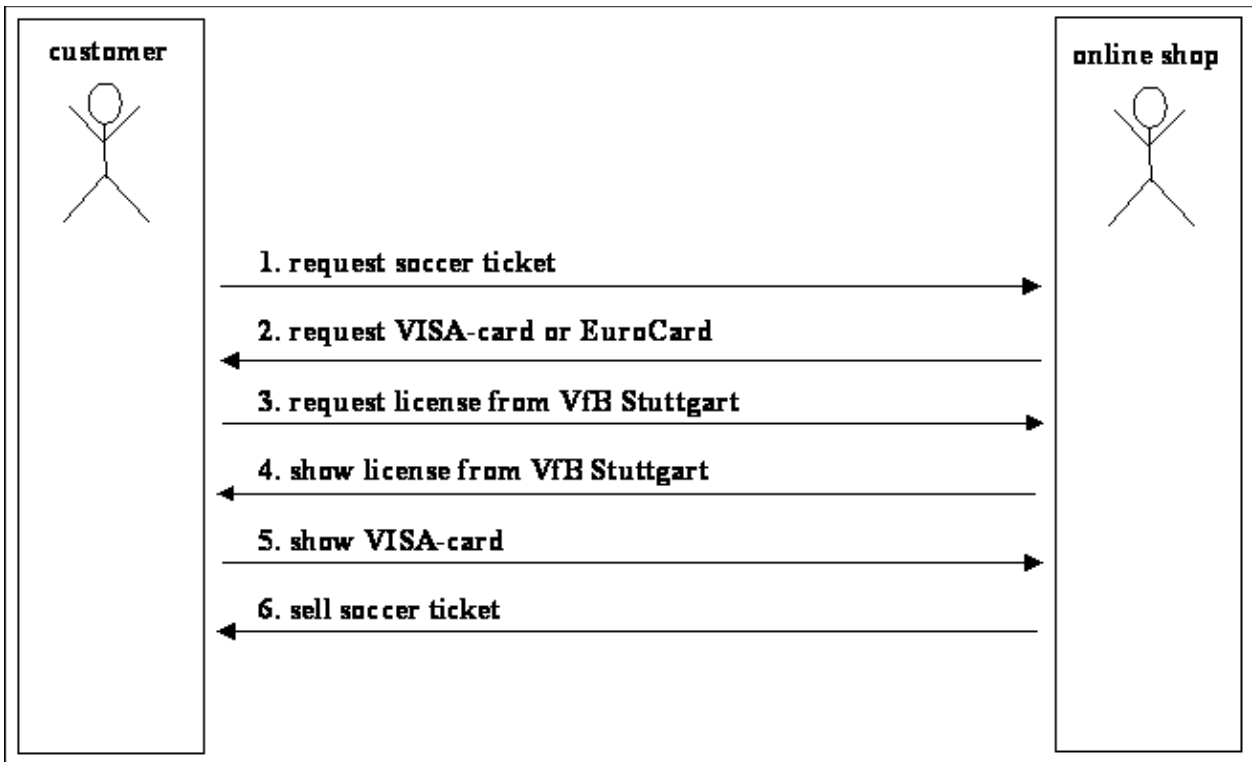
and the following sensitive resource:

- *ticket for soccer game (protected by policy2)*

and the following policy:

- *policy2:*
  sell ticket only, when the customer presents a VISA-card-credential signed by VISA or an EuroCard-credential signed by EuroCard

The negotiation agents of the two parties behave as the following image illustrates to fulfill the transaction:



1. The customer accesses the online shop and wants to buy a ticket for a soccer game of his favorite club VfB Stuttgart.
2. At the online shop, the ticket is protected by a policy that says that the customer must present either a VISA-card- or a EuroCard-credential to the online shop. This policy is disclosed to the customer.
3. The negotiation software of the customer processes the received policy and finds out that the VISA-card-credential is available, but protected by a special policy. The customer created it, because he has heard about companies that cheat by selling tickets, so he only wants to buy them at a company that can show a license signed by the VfB Stuttgart in form of a special license-credential. This policy is disclosed to the shop.
4. The negotiation agent of the online shop processes this policy and discovers that the license-credential required is available and not protected by any policy, so he sends it to the customer.
5. The policy that the agent of the customer sent to the shop is satisfied by the received license-credential. Now, the protected VISA-card-credential that was requested by the policy of the shop can be send to him.
6. The agent of the shop receives the VISA-card-credential and finds out that the policy that protects the sensitive resource, the soccer ticket, is satisfied. Now the customer can be billed and the ticket can be send to him.

2.5.2 Second Example

The second example is about a person who studies at the university in Hannover and wants to access an online service that shows job offers exclusively for students in Hannover and can't be used from other users.

The **student** has the following credentials:

- *identity-credential which contains information (address, name, ...) about the student (signed by a municipal authority in Hannover, protected by no policy)*
- *income-tax-card-credential (signed by a municipal authority in Hannover, protected by no policy)*

and no policies.

The **student job service** in Hannover has no credentials

and the following sensitive resource:

- *website with job offers for students (protected by policy1 and policy 2)*

and the following policies:

- *policy1:*
  present web site with job offers only when the requester shows a student-credential that is signed by the university in Hannover
- *policy2:*
  present web site with job offers only when the requester shows a income-tax-card-credential signed by a municipal authority in Hannover

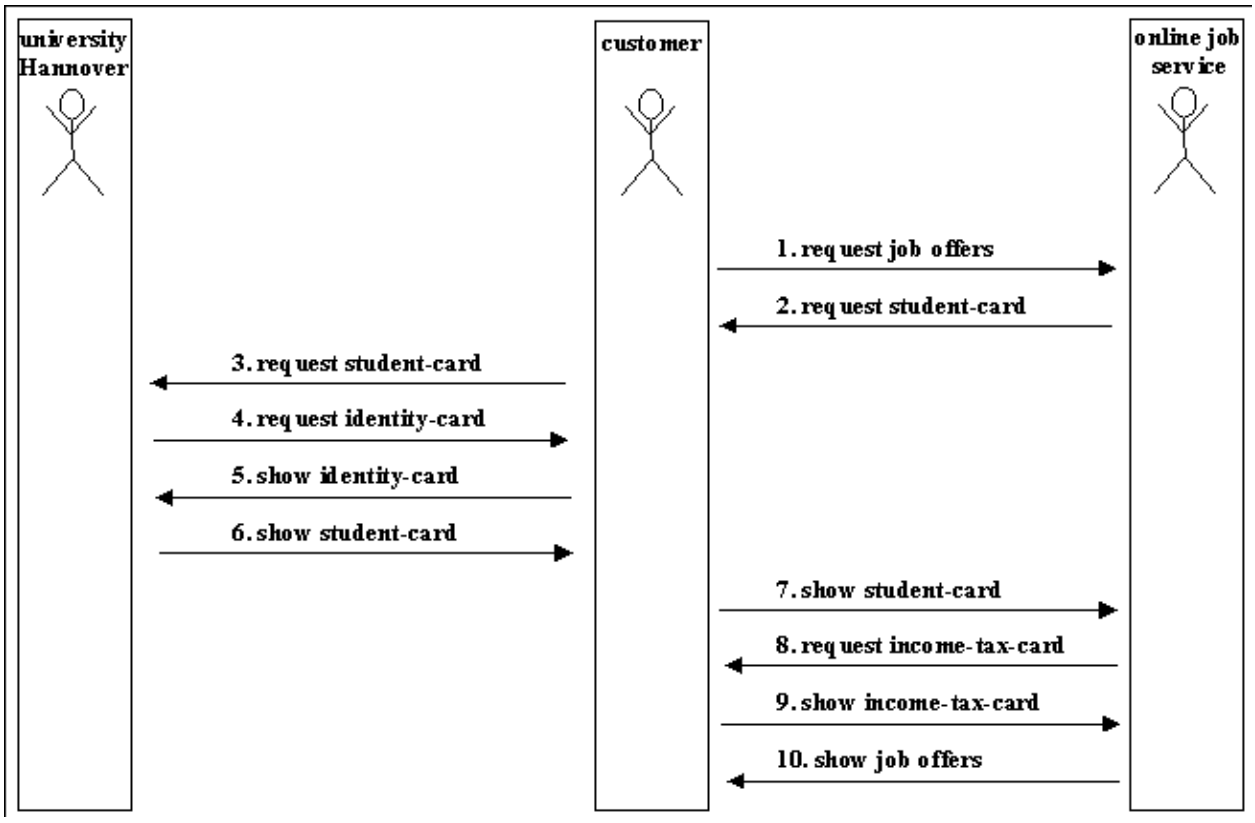
The **university in Hannover** has the following credential:

- *student-credential (signed by university in Hannover, protected by policy3)*

and the following policy:

- *policy3:*
  show student-credential only to requester, if he presents a matching identity-certificate signed by a municipal authority in Hannover

The negotiation agents of the three parties behave as the following image illustrates to fulfill the transaction:



1. The student accesses the online service and wants to see the website with job offers that only students of the university Hannover can read.
2. At the online service, the website with job offers is protected by a policy that forces the requester to show a student-credential signed by the university in Hannover that proves if the requester really studies there. This policy is disclosed to the student.
3. The student receives the policy and his negotiation agent discovers that he doesn't have the student-credential, so he requests the university in Hannover for it, because it is the issuer of the credential.
4. The agent of the university in Hannover receives the request, but the student-credential is protected by a policy that wants the requester to show an identity-credential signed by a municipal authority in Hannover to find out if the requester is really registered at the university in Hannover. This policy is disclosed.
5. The agent of the student receives the policy and sends back the identity-credential to the

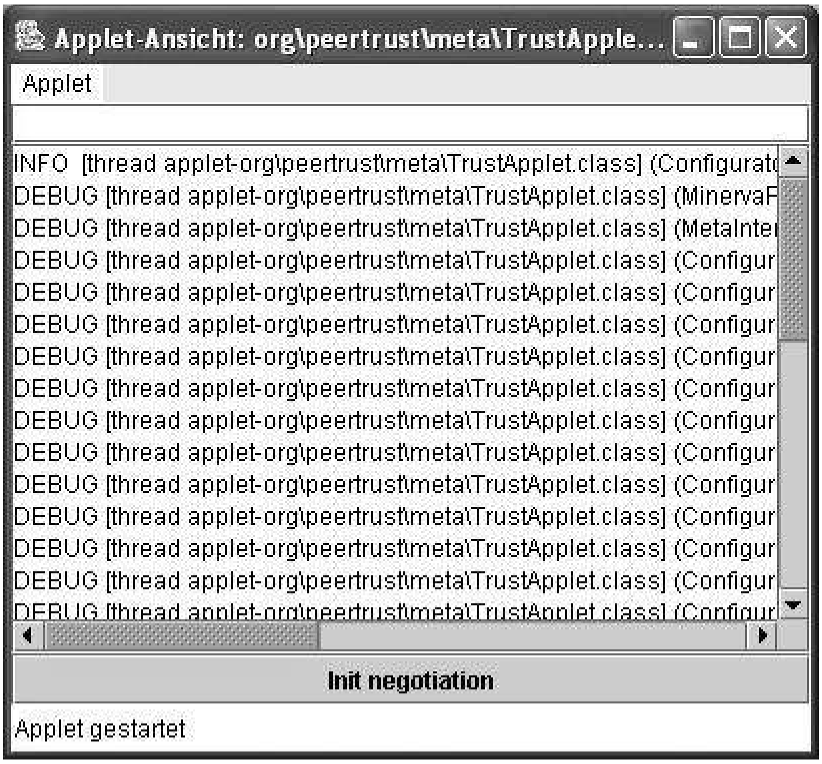
university as it is not protected by any policy.

6. The agent of the university in Hannover gets the identity-credential, now the policy that protects the student-credential is fulfilled and the matching student-credential is send back to the student.
7. After receiving the student-credential, the agent sends it to the online service, because he requested it before.
8. The agent of the online service receives the student-credential and the first policy that protects the website with job offers is satisfied. But the site is also protected by another policy that wants the requester to present an income-tax-card-credential signed by a municipal authority in Hannover. This policy is disclosed to the student.
9. After receiving the policy, the agent of the student discovers that the matching income-tax-card-credential is available and not protected by any policy, so he sends it to the online service.
10. The agent of the online service receives the credential and the second policy that protected the website with job offers is satisfied. Now access to that site is granted to the student.

Currently there are a lot of trust-negotiation systems available beside PeerTrust, to which the whole next chapter is about. Also a small description of the concurrent systems and a comparison to PeerTrust can be found there.

3. PeerTrust

This chapter is dedicated to PeerTrust [2,3], an automatic trust-negotiation system that contains all the features and concepts that were described last chapter in context with trust negotiation. PeerTrust is written in Java and can be delivered as an application or as a signed applet (see the screenshot below) to the peers that want to use it in order to be able to automatically negotiate trust with other parties who also use PeerTrust. This software is still in a prototype-phase and is open source (http://sourceforge.net/projects/peertrust/). This chapter will describe the functionality, features, important algorithms and other relevant topics of PeerTrust, which are the basics on what the security-enhancements of this Bachelor-thesis build up that will be investigated from the next chapter on.



It was already described how important a language for a trust-negotiation system is, as it has to be expressive enough and at the same time not too complicated, so that even amateurs and people who have little experience in this topic are able to use it in order to create their own policies and rules (maybe with special tools helping them). The following text will now focus on the syntax and semantics of the PeerTrust-language and point out the special features that other languages of trust-negotiation systems don't have.

3.1 Syntax of the PeerTrust-language

The origin of the syntax of the PeerTrust-language [2,3] comes from first order Horn rules (definite Horn clauses which are the basis for logic programs that have been used for the rule layer of the Semantic Web), which look this:

lit_0 <- lit_1 , ... , lit_n

In detail, *lit_i* stands for a positive literal *P_j(t_1, ... , t_n)*, where *P_j* is a predicate symbol and the *t_i* are its arguments, which are themselves terms (function symbols and its arguments, which are again terms). *lit_0* is also referred as head of the rule, and *lit_1* to *lit_n* (a conjunction of literals) is called the body of the rule, which may also be empty. The head of the rule is true, when the body is true. For example, in the rule

f(X) <- g(X) , h(X)

f(X) is true, if both *g(X)* and *h(X)* are true. As the body consists of a conjunction of literals, the user has to split a rule in several rules when he wants use a disjunction. Rewriting the example above as a disjunction (f is true, if at least g or h is true) would lead to:

f(X) <- g(X)
f(X) <- h(X)

When the body of a rule is empty, the head is always true, this rule is also called a fact then.

3.1.1 The Issuer-argument

One of the features of the PeerTrust-language syntax that other trust-negotiation systems don't have is the ability to specify for each literal which party must process it. This way, a precise delegation of evaluation is possible. Each party can write more powerful rules or policies with this delegation-feature, and when having to evaluate a rule, the software can easily determine if a literal must be processed locally of delegated to another peer. To express such a delegation of a literal *lit*, it has to be extended by a special argument *Issuer* (specifies, which party or authority must evaluate *lit*), written as follows:

lit @ Issuer

For instance a Party has to process the following rule:

f(X) <- g(X) @ steve, h(X), i(X) @ myra

His negotiation software discovers, that only *h(X)* has to be evaluated locally, because the *Issuer*-extension is missing here. This is not the case in the two remaining literals of the rule's body, so *g (X)* is delegated to a party named *steve* and *i(X)* to party *myra*. The software then waits for the answer of these two delegation-partners in order to continue processing the rule.

A literal *lit* is not restricted to have only one *Issuer*-argument, he may have several of them, this would be written as:

lit @ Issuer_1 @ ... @ Issuer_n

The outer *Issuer*-argument is evaluated first. This leads to more variety in delegation. For instance, in the example in the last chapter, where the student wants to get access to a website with job offers, the online service requests a student-card-credential. Delegating the request for this credential directly to the university it not a good idea, because it will certainly refuse requests that doesn't come directly from their students, due to protection of data. Instead, the web service will request the student to present his student-card-credential that is signed by the university, the relevant policy may look like this:

access(job_list , Customer) <- student_card(Customer) @ university_hannover @ Customer

The *student_card*-predicate is evaluated to true, if the student-card-credential is shown. In PeerTrust, literals may represent external procedure calls to get information from the other party (like time or date) or access special libraries. Another possibility is to use these external calls for authentication, in the case of the *student_card*-literal, the negotiation software is forced to find and present the appropriate credential. Additionally, the order of the *Issuer*-arguments are important. As the outer *Issuer* is evaluated first, the student has to present the matching credential. There are two possible cases: if he possesses it, his negotiation software will present it to the online service directly if it is not protected by any policies. If the student doesn't have the credential, he must ask the university for it, this is what the inner *Issuer*-argument is for. So he delegates the literal to the university and receives the credential, either immediately or after a few negotiation steps, depending on the policies that protect it. The negotiation software may store it locally, because if another party will request it, too, he will not have to ask the university for it again. That's the intention of a sequence of *Issuer*-arguments: if the outer *Issuer* can't evaluate the literal, he can delegate it to the party corresponding to the left *Issuer*-argument and so on. How exactly the

credentials look like, how they are implemented, disclosed and verified by the other party will be described later (in this and following chapters, as the tasks in this Bachelor-thesis had also something to do with these special topics).

Another possible use case for delegation is also given when a party wants to participate in a trust negotiation, but his computing device doesn't fit the requirements, like processing power or a certain amount of memory, in order to negotiate trust. This may be the case if a party uses a handheld device (PDA or (smart- or mobile) phone). The negotiation can be delegated to another party which is able to do it and the handheld user just waits for the results that are sent back. When taking the future into account, such methods may be used less, as phones and PDAs become more and more powerful. They will be certainly, maybe in some cases already are, able to negotiate trust itself, without any help.

3.1.2 The Requester-argument

When evaluating a literal, a party should also know which peer has asked for or has delegated it because this information may also affect the evaluation. Imagine the last example in the previous chapter where the university only presents the student-card-credential, if the party which requested it is a student. Although the university can find it out by protecting the credential with a policy that forces the requester to show another credential (for example a digital identity-card), the university can also work with a database of all peers of his students. When receiving the query for a student-card-credential, the negotiation software knows which peer sends it and asks the database if this one belongs to a student.

The exact syntax for finding out which peer has requested for or has delegated a literal *lit* is:

lit $ Requester

or in combination with an *Issuer*-argument:

lit @ Issuer $ Requester

A rule that corresponds to the case described above (if a requester asks the university for a student-card-credential, it will only be presented if the requester peer has an entry in a special database where all the students of the university are stored) may look like this, when using the *Issuer*-argument:

access(student_card(Student)) $ Student <- is_included_in_database(Student)

Also this argument can be used more than once for a literal, like the *Issuer*-argument. If more than one *Requester* is specified this way, this can be seen as a sort of history of requesters with the most recent one appearing as last *Requester*-argument. This way, not only the name of the last requester can be accessed in a rule, also the ones that delegated the literal to him. Coming back to the example with the student and the online service that offers jobs just for students, the university that owns the student-card-credentials may decide to change the protecting policy, because now it has a cooperation with the online service. Therefore, the student-card-credentials are only presented to students, if their peer appears in the database and, additionally, the literal was delegated to them from the online service. The new, modified policy may look like this:

access(student_card(Student)) $ online_job_service_hannover $ Student <-
is_included_in_Database(Student)

3.1.3 Signed and normal rules

At the beginning of this chapter, the relation of the PeerTrust- and definite Horn clauses-rules was described. Together with the two additional arguments explained above (*Issuer* and *Requester*), the user is able to write powerful rules and policies to protect his sensitive resources. But in PeerTrust, it is also possible to cache and use rules from other parties. These ones differ from the (local) rules and policies that the user has written, because of security reasons. The user may manipulate them, therefore these rules that he has obtained from other parties need to be signed by those. How this signing works and the other party can verify if the signed rule is valid, is part of the next chapter. The syntax of local and signed rules differs, signed ones contain an *Issuer*-argument and a *signedBy*-statement, followed by the name of the *Issuer* in brackets. So, for example, when regarding the following facts of a knowledge base:

f(X)
h(X) @ amanda signedBy [amanda]

f(X) is a local and *h(X)* a signed rule. Coming back to the example with the student, university and online service, the university may not itself present the student-card-credentials, instead it may delegate this to a special inner department named *universityStudentDepartment*. When the user gets back the credential, it is signed by the department, not by the university itself, as the online service requests. A signed rule can help here. The department can also transfer an additional signed rule like the following to the student:

student_card(Student) @ university <- student_card(Student) @ universityStudentDepartment signedBy [university]

The student then can pass both the credential and the signed rule to the online service in order to convince it that the credential is valid although it is not issued by the university.

3.1.4 Public and private policies/rules

A useful feature that many object-oriented like Java or C++ have is the possibility to protect methods and attributes of classes from being accessed from an outer class or method. The same concept can be transferred to trust negotiation, instead of methods and attributes, predicates are concerned. These may either be public or private. Public predicates can be queried from other peers, they may be disclosed. In contrast to that, this isn't possible with predicates that are marked as private, they can't be called from or shown to other parties. Examples for such predicates are such ones that represent external function calls (as discussed earlier in this chapter), for instance some that are responsible for authentication or some that return information from the underlying platform on which they run like date and time. Public and private rules are easy to implement in definite Horn clauses and because the PeerTrust-language is based on that, they are also possible there.

3.1.5 Guards

Another possibility that the PeerTrust-language-syntax offers is to specify the evaluation order of the different literals in the body of a rule. As already explained earlier, trust can be negotiated incrementally by forcing the other client to show credentials at the beginning that are not so sensitive and continue to request more and more sensitive ones while the negotiation proceeds. So, the requested party has to pass several small tests which may demand more and more private information in order to get access to the desired resource. Instead of specifying a lot of policies to achieve the behavior described, this can also be done with just a single one. The syntax offers a special operator (|) for determing the order in which the literals must be evaluated. Compared to the negotiation concept mentioned above, literals that force the other party to reveal not so sensitive information must be evaluated at first, whereas the credentials that must be presented later become more and more sensitive. This is just one use-case for the operator, he generally increases the variety of the PeerTrust-language and can help to decrease the number of rules and policies. Both signed and local rules can contain this operator, a local rule with it is written as follows:

lit_0 <- lit_1 | lit_2

of course, the operator can be used more than once, for example:

lit_0 <- lit_1 | ... | lit_n

The |-operator divides the body of a rule in different sets. Before the literals in a set can be processed, all literals in the previous one must be evaluated to true. The leftmost set is processed first. All but the last set are also referred as guards. Regarding the rule specified above, the party that has to process it must first evaluate the first set (literal *lit_1)* to true before continuing with the next one. *lit_n* is the last set and can only be processed, if all previous sets were successfully evaluated.

When looking at the example with the student, the university and the online service with job offers (as it was specified at the end of the previous chapter), it is obvious that the sensitive resource of the online service (the website with job offers just for students) is protected by two policies. With the help of the new operator, we can transform these two into a single policy. The evaluation order will not be different from the previous version, because the user has to show a student-card-credential first and then an income-tax-card one. The new policy may look like:

access(job_list , Customer) <- student_card(Customer) @ university_hannover @ Customer | income_tax_card(Customer) @ municipal_authority_hannover @ Customer

3.2 Semantics of the PeerTrust-language

According to the semantics of the PeerTrust-language, they represent an extension of SD3 [8,16]. For each literal, the *Issuer*- and *Requester*-parameters, as well information about distributed *peers* and their knowledge base, is combined with the ordinary parameters of the predicate for internal representation. If the *Issuer*-argument is not specified, the Self-variable is used for it, which stands for the local peer. So, when the user writes predicate in a rule or policy as follows:

P(t_1 , ... , t_n) @ Issuer $ Requester

it is stored and processed locally as:

P_+(t_1, ... , t_n , Issuer , Requester , Peer)

The Issuer- and *Requester-* parameters of this internal storage-form can contain more than one entry, but the description of how exactly the last three parameters are specified won't contribute much to this Bachelor-thesis, so is omitted.

3.3 Binding policies to resources

The binding of policies to sensitive resources (for example if the trust negotiation software has to find out what policies a user has to satisfy and how this can be done before getting access) isn't done directly in PeerTrust. Instead, as this fits better to the Semantic Web (PeerTrust can also be used in it), the policies are assigned to the sensitive resource that they should protect with the help of the RDF (Resource Description Framework)-metadata of it. Subsequently this metadata is imported to the automatic trust negotiation software as rules and facts of a logic program. These rules may contain special predicates and restrictions (like time or date) in order to get as much information as possible out of the metadata. If these satisfy the policies that protect the resource, they are fulfilled and the user can gain access to the resource.

As an example, here is the RDF-metadata of a certain document that author -max man has written:

<rdf:Description about=http://www.xyz.de/.../documents>
<rdf:type resource=http://www.xyz.de/.../documents/document_a.#Document>
<dc:creator>
max man
</dc_creator>

The following policy is protecting the document:

read(Document)$Requester <- dcCreator(Document,Requester)

When the automatic negotiation software combines the content of the RDF-metadata and the assigned policy, it finds out, that -max man is the author (*dcCreator*) of this document, so the policy says that he is the only one that is allowed to read it.

3.4 The PeerTrust-program

As already mentioned, PeerTrust [2,3] can be delivered as a Java application or signed applet, the

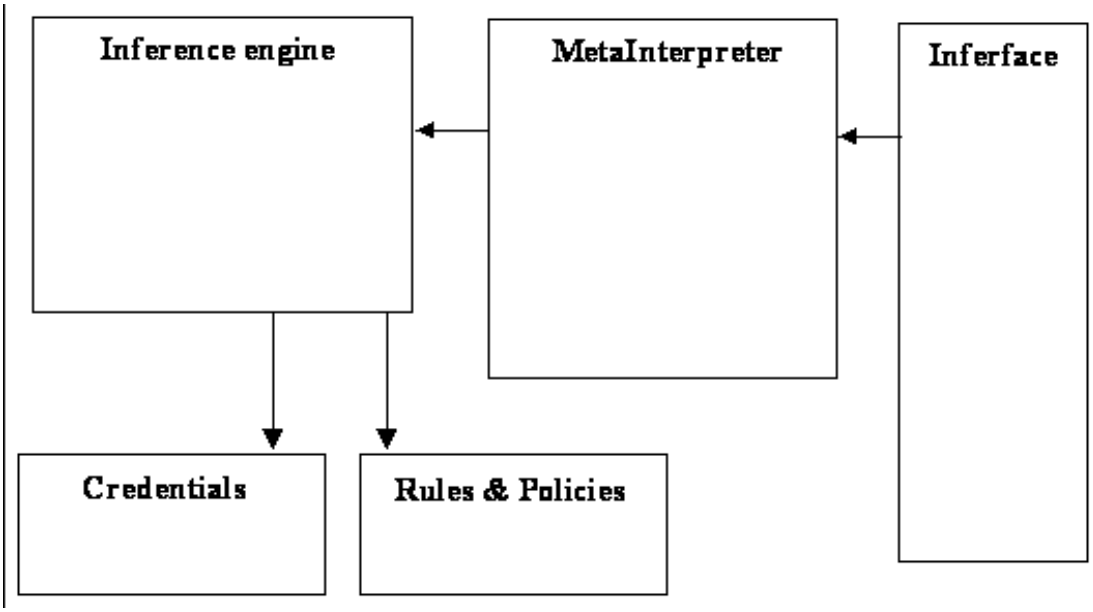
latter one has the advantage that it can be automatically downloaded and started by using ordinary internet-browsers like Microsoft Internet Explorer, Netscape Navigator, Mozilla or Opera, because of the built-in Java support for applets. As the PeerTrust-Java-applet is signed, the user can determine who has signed it by looking at the certificate chain included. Automatic trust negotiation is a sensitive task, so in order to determine if the software works as it should, without any hidden parts that spy out or harm the user, as it is done by virus- or dialer-software, the applet should be signed by an organization that is well known and that can be trusted. A description of certificates and certificate chains will appear later, as it is an important part of the next chapter. So, depending on the settings of the browser used, either it or the user himself decides whether to start the PeerTrust-applet. Alternatively, the applet may also be started without any browser, instead some other programs can be used for that like the appletviewer-program that is shipped with every (Standard- or Enterprise) Java-SDK (Software Development Kit) by Sun. As normal Java applets have only limited access to the computer on which they run, because of security reasons, this is not sufficient for PeerTrust. Operations that affect security like network- or I/O-access are not possible in applets by default, special permissions must be given to the applet in order to activate those features (a special tool that is shipped with the Java-SDK named policytool can be used for that special purpose). This is also done with the PeerTrust-applet and is another reason for the fact that it is signed.

More information about PeerTrust and downloads (examples, metainterpreters and traces that help to understand how the evaluation algorithm works) can be found in the internet:

http://www.learninglab.de/english/projects/peertrust.html
http://sourceforge.net/projects/peertrust/

3.4.1 The structure of PeerTrust

The following text will now concentrate more on the PeerTrust-software itself and its individual parts. The following figure illustrates this:

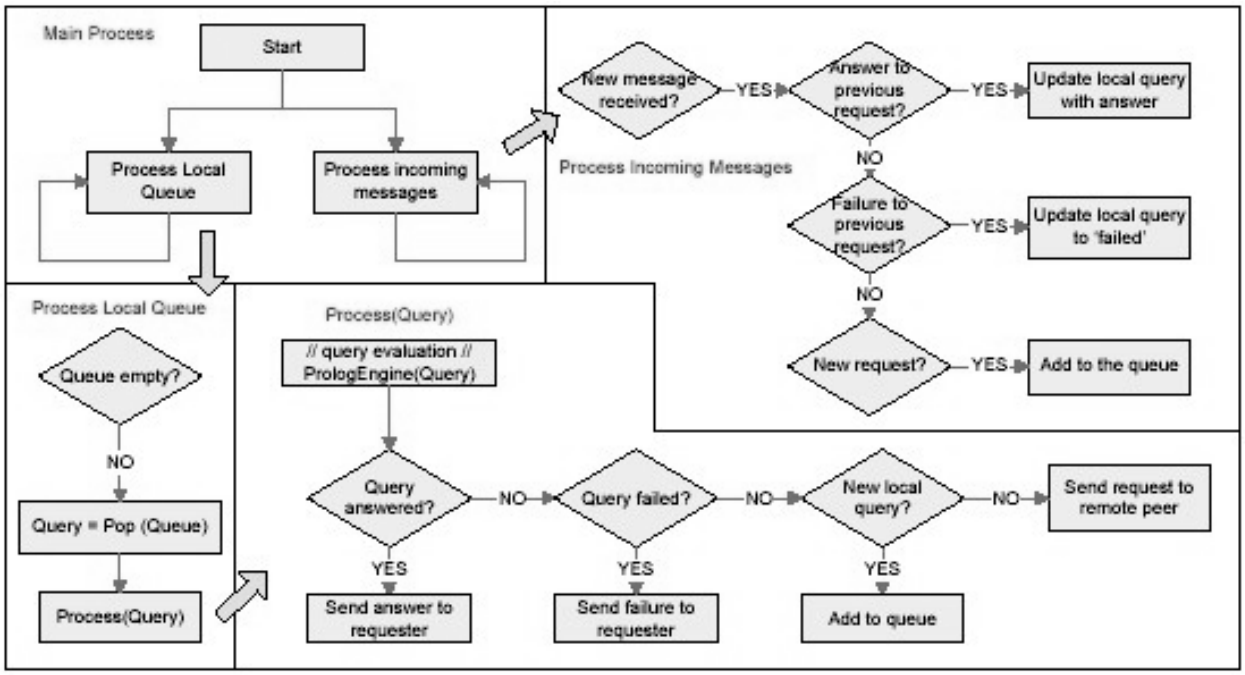


The different parts basically do the following (a more detailed description of some of them listed below can be found afterwards):

- The **Interface** is responsible for communication with other users who can only directly access this part.
- The **MetaInterpreter** receives checks and processes incoming queries or answers to previous sent queries from the Interface. He also sends queries or answers to other parties.
- The **Inference engine** has access to all local rules, policies, metadata (written in RDF, Resource Description Framework) and credentials (X.509 credentials with extensions, these will be described later as they play a major part in the following chapter) and can draw inferences from them. The MetaInterpreter processes queries with the help of this part. The inference engine uses internally the Prolog-implementation of Minerva, as it offers libraries for Java.

3.4.2 MetaInterpreter and evaluation algorithm

The MetaInterpreter is one of the most important parts of PeerTrust, as it covers the processing, delegation, querying and disclosure of policies and rules, as well the underlying negotiation strategy and mechanism. The exact behavior of the MetaInterpreter will now be investigated, its algorithm that is responsible for evaluating rules is illustrated on the figure below (taken from [2]).



The MetaInterpreter uses a queue, currently a FIFO (first in first out) one. It stores all the queries and subqueries that have to be evaluated. In order to increase security, each of these (sub-) queries are contained in a special Tree-object that also possesses a proof tree that has all the necessary information (rules used, instantiations of variables, credentials, ...) to prove that the query is satisfied (therefore it has to be updated continuously, also during delegation), as well a list of all the subqueries that can be expanded. Proof trees will be described more detailed in the next chapter, with them, parties that receive answers to queries they sent before are able to find out if the answer is valid and really matches to the query sent. The terms “Tree-object” and “query” will be used further in this document to describe the same thing. Certain predicates can't be evaluated locally and therefore have to be delegated to third parties. The *Issuer*-argument specifies the party to which it must be sent. So the delegation of queries, the processing of received answers to previously sent ones, as well the transmission of answers to queries that came from a third party have to be considered in the evaluation algorithm of the MetaInterpreter. Each Tree-object has a status, one for a negative (“failed”) and two for positive evaluations (as there may be many possible answers to a single query, the last answer has a different status (“last answer”) than the possible answers before “answer”)) and one for the case that a query hasn't been processed yet (“ready”). If a query is delegated to another party, the status of the local one is set to a special status “waiting”, if an answer (that is not the last possible one) is returned, this status is set to “answered and waiting”. The waiting-states indicate that the query is waiting for answers, so it doesn't need to be evaluated locally, other queries in the queue can be processed meanwhile. For each query (both the ones that came from other parties and the ones that are sent to others) exists a Tree-object in the queue. If new subqueries are expanded, also new Tree-objects for

these are created and added to the queue. Answers to previously sent queries are combined with the corresponding local Tree-object in the queue (for instance the proof tree in it has to be updated).

Coming back to the illustration, the MetaInterpreter uses two different threads in order to evaluate and delegate the queries and rules. The first thread, to which the left and lower part of the illustration refer, is responsible for processing (the (sub-) queries included in) Tree-objects in the queue. He always gets the (sub-) query that is stored in the Tree-object in first position of the queue. If this query is satisfied, the (positive) answer will be sent to the requester, if it failed, the (negative) one will be sent to the requester. Only (sub-) queries that were answered are deleted from the queue, because there is really no need to process them any further. If those cases described above don't match, the query is evaluated locally (its subqueries are expanded) and the results are examined, if one can be processed locally, it will be added to the queue again, if not, it is delegated to the suitable party as a query. The local query is set to some special state for waiting, as already described in the previous paragraph.

The algorithm of the first thread (it is called in an endless loop) written in pseudo-code:

```
get query from queue
if the queue contains no query {
        sleep for some milliseconds
        return;
}
if the evaluation of the query has failed {
        send negative answer to the party that sent the corresponding query
        return;
}
if the query is ready (not waiting, answered or failed) and has no more subqueries or it is
        answered {
        send positive answer to the party that sent the corresponding query
        return;
}
process the query with the inference engine
if no answers can be found to that query {
        if there is no other query pending from the same requester {
                create copy with status set to failed
                add copy to the queue
        }
```

```
}
else {
        iterate through all the found answers {
                if answer must not be delegated to another party {
                        if answer has no remaining subgoals {
                                create copy with status set to answered
                                add copy to the queue
                        }
                        else {
                                create copy
                                add copy to queue
                        }
                }
                else {
                        create copy as local query with status set to waiting
                        add copy to the queue
                        send new query to specified party
                }
        }
}
```

The second thread is responsible for processing and handling incoming messages (for instance queries and answers). If the thread detects that a new message arrives, it looks if it is the positive answer to a query that the other thread had sent previously. If that is the case (either a normal or a last answer), the corresponding query is updated with the answer, also its proof tree, if not, the thread will investigate if the received message represents the negative answer to a previously sent query. If that is the case, the suitable query will be marked as failed, it doesn't have to be processed any more, if not, the message must be a query from another party. Then, the query is added to the queue, so that the other thread can evaluate (or delegate) it the way it is described in the paragraph above and can send the result back afterwards. If the other party doesn't return an answer to the previously sent query in a special amount of time, this is also reported to the local query.

The algorithm of the second thread (it is called in an endless loop) written in pseudo-code:

```
get received message
if message is a query from another party {
        create copy of query
```

```
        add copy to queue, the algorithm above is responsible for evaluating it
}
else if message is a (positive) answer to a previously sent query {
        search for local query that matches this answer
        unify this query with this answer
        create a copy of the query
        add the proof tree from the answer to the copy
        add the copy to the queue
        if the status of the query is -waiting {
                set status of the query to -waiting and answered
                update the queue
        }
}
else if message is the last (positive) answer to a previously sent query {
        search for local query that matches this answer and delete it from the queue
        unify this query with the answer
        create a copy of the query
        add the proof from the answer of the copy
        add the copy to the queue
}
else if message is a (negative) answer to a previous sent query {
        search for local query that matches this answer
        if the query has received no other query before {
                set the status of the query to -failed
                update the queue
        }
        else {
                remove the query from the queue
        }
}
```

These algorithms are already integrated in the current PeerTrust-prototype. As already mentioned, for inferencing and evaluating rules or policies, the Minerva Prolog-implementation is used (as it offers special libraries that can be accessed from Java and is portable). The communication with other parties is done over secure sockets (with SSL/TLS), X.509 certificates are used for authentication and credentials in PeerTrust. All these three topics will be covered in the next chapter, because the tasks in this Bachelor-thesis had to do with them.

3.5 Related Work to PeerTrust

The secure Dynamically Distributed Datalog (SD3) [8] offers a policy language which allow users to write their own high level security policies. The evaluation of them and verification of credentials is done by SD3. The language of SD3 (based on Datalog) is extended by the PeerTrust-language, so when comparing these two, the latter one is more powerful. Although SD3 offers an argument similar to the *Issuer*-argument of PeerTrust, you can't specify more than one per predicate as it is possible in the PeerTrust-language. Equivalents to the *Requester*-arguments and guards don't exist in the SD3-policy language. What is also missing in contrast to PeerTrust is the ability to keep policies private in order to restrict them to be only used locally and not be disclosed to other parties. A newly proposed policy language named “Cassandra” [9] overcomes those restrictions as it has adapted many features of the PeerTrust-language and is therefore more close to it.

SSL (Secure Socket Layer) and TLS (Transport Layer Security) [5,6,14] are widely known and accepted protocol-standards for authentication and exchanging credentials and certificates in the web. Trust negotiation can't be done with SSL/TLS directly, special extensions must me made for that [6], for example the client must be able to disclose his policies to the server, which isn't possible in unmodified TLS. PeerTrust uses SSL/TLS only for secure socket communication, the exchange of credentials is done manually without using the mechanisms of SSL/TLS. A closer look to TLS will appear in the next chapter.

Yu et al. [10] concentrate on autonomy and privacy in automatic trust negotiation. In their approach, parties must agree on a special strategy set/family before the negotiation of trust begins. As already explained in the previous chapter, limiting the trust negotiation software to use only one, fixed strategy isn't appropriate, instead the software/client should choose a special strategy that specifies what should be disclosed next. As different strategies may not cooperate, strategies are divided into special strategy sets/families which are proven to work together, so parties that want to negotiate trust must agree on a strategy set/family first. Yu et al. have found a solution for that, the current PeerTrust-prototype restricts the parties to use only one strategy, but support for more autonomy is planned for future versions.

Li et al. [11,12] work on role-based languages for trust negotiation, roles described in credentials can be mapped to the matching identities this way. Delegation of evaluation works similar as in the PeerTrust-language. The language of Li et al. also offers acknowledgement policies to determine in which order a policy must be evaluated in order to prevent that other parties can get all the local policies and the information included in them (for instance about credentials they should protect). PeerTrust has a similar feature called guards, an operator divides the predicates in order to specify the order in which they should be evaluated.

3.6 Further Work on PeerTrust

For PeerTrust 2.0, a lot of improvements and new features are planned. For instance, the ability to export policies to and import them from RuleML format, or support of XML digital signatures.

As already mentioned above, more autonomy is planned. In the current prototype, only one strategy of disclosing policies or credentials is possible. Trust negotiation software ideally should support more strategies in order to increase autonomy and flexibility, existing systems already contain this feature. Another idea is that the software analyses if the trust negotiation can succeed and take this information in consideration when choosing a strategy or disclosing policies or credentials (for example he can discover the consequences if he refuses to present a credential). But these concepts lead to more complexity, the (relatively simple) MetaInterpreter of the current PeerTrust-prototype (and its algorithm) isn't sufficient any more to complete these tasks. Yu et al. [10] have done research on these topics, their results can be taken in consideration when implementing those features in PeerTrust.

Other policy languages already offer the ability to write policies that offer several types of access to a resource and intensional specification, this is done by content-triggered variety of trust negotiation at run time. For instance, the ability to read all documents that were written yesterday and whose language is German. In contrast to that, in the current PeerTrust-prototype, policies are restricted to protect only one policy and one type of access, the integration of content-triggered trust negotiation is planned, and it should also work correctly in the Semantic Web.

4. Extending PeerTrust

This chapter will describe the tasks that had to be solved in this Bachelor-thesis. Their aim was to increase security in the current PeerTrust-prototype. Trust negotiation may be used in very sensitive areas like online shopping or online banking, so security plays a key role when users decide to use trust negotiation software or not, nobody wants his own credentials to be misused or access to sensitive local resources is gained by unknown parties like attackers. Some methods to hinder hackers from manipulating a trust negotiation are presented in this chapter.

The work in this Bachelor-thesis consisted on the three following tasks which are described in more detail later:

The first task was the implementation of the authenticatesTo-predicate which is meant for additional authentication. A user may have several identities (based on the certificates he owns), this predicate can ask him to present a specific one which must also be signed by a specific authority. By using this predicate in policies, security can be increased by forcing the other party to authenticate himself more than once and prove that he possesses certificates that match to specified identities and authorities that are parameters of the predicate. The handling of this special predicate and the exchange of the suitable certificate (chain) (description will follow later) is done manually, without existing solutions like TLS that were used in the previous approach.

In the second task, the implementation of the verification of the proof tree that is contained in every Tree-object (represents a query that must be evaluated, as already explained in the previous chapter) had to be done. When a peer sends a query to another party and receives the answer, the proof tree enables him to find out if the answer fits to the query he sent and if the answer can be logically inferenced from the contents of the proof tree (which consists of rules used, variable instantiations and credentials). If this doesn't succeed, the answer will be rejected and marked as being failed. That is basically what the algorithm does (with help of the Minerva Prolog inference engine), beside checking if the signed rules and the corresponding credentials are alright. The proof tree representation in the Tree-class had also to be changed for that as the previous existing one was not sufficient for signed rules and credentials.

The third task was the easiest one which was about implementing the verification of credentials. As these are contained in certificates, both ones had to be checked, the Java SDK contains a lot of helpful methods for that. The solution of this task is also used by the previous one (as the credentials in a proof tree also need to be checked there).

4.1 authenticatesTo-predicate

In [1], the authenticatesTo-predicate was introduced. The exact syntax is:

authenticatesTo(Identity,Party,Authority)@Requester

This predicate is evaluated to true, if *Requester* can proof to *Party* that he possesses the identity *Identity* issued by authority *Authority*. If *Identity* isn't specified, *Requester* should provide the matching name for it in the case the predicate succeeds. For example if peer -jules wants peer -mary to prove that she has the identity -maryanne issued by authority -veriSign, the predicate would look like this:

authenticatesTo(maryanne,jules,veriSign)@mary

As mentioned in [1], there are a lot of possible interpretations for the authenticatesTo-predicate, for example logins or biometric tests, but this Bachelor-thesis will focus on the use of X.509 certificates here, because this concept is already being used in the current PeerTrust-prototype. A description of X.509 certificates and certificate chains can be found below.

Security can be increased with this predicate (implemented with the help of X.509 certificates), because additional authentications lead to more protection against hackers. Instead of having to present just one certificate (chain), multiple ones that differ from another may be asked for. An attacker must get access to nearly all certificates/certificate chains, not just a single one, a party has in order to authenticate as the other party and manipulate the trust negotiation this way. To make it more difficult, just presenting the right certificate (chain) isn't enough, the party must also prove that it owns the private key corresponding to it, to convince the other party that he is really the owner. Getting access to all certificates/certificate chains and private keys of another party is very hard for an attacker, as those are in most cases also protected by passwords.

4.1.1 X.509 certificates and certificate chains

X.509 certificates [4,5,13] are suitable to be used for authentication (and so also for the authenticatesTo-predicate). X.509 is a common format for certificates that is well-known and widely used, for example for authentication, the Internet, electronic mail or other functionality in security systems. It is currently in its third version which was released 1996.

This format is specified as followed (some important entries are described below) :

```
Certificate  ::=  SEQUENCE  {
        tbsCertificate       TBSCertificate,
        signatureAlgorithm   AlgorithmIdentifier,
        signatureValue       BIT STRING
}

TBSCertificate  ::=  SEQUENCE  {
        version         [0]  EXPLICIT Version DEFAULT v1,
        serialNumber         CertificateSerialNumber,
        signature            AlgorithmIdentifier,
        issuer               Name,
        validity             Validity,
        subject              Name,
        subjectPublicKeyInfo SubjectPublicKeyInfo,
        issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                -- If present, version shall be v2 or v3
        subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                -- If present, version shall be v2 or v3
        extensions      [3]  EXPLICIT Extensions OPTIONAL
                -- If present, version shall be v3
}
```

These certificates offer entries for information (name, organization, alias-name, ...) for both the identity for which this certificate is meant for (the **subject**), as well for the identity of the one who issued it (the **issuer**). Although these two entries may be equal (a person has made a certificate for himself, in this case it is called a -self-signed certificate), the issuer of a typical certificate is different from the subject.

Of course, there exist more entries than just the ones for the subject and issuer, that wouldn't be enough to enable parties to verify and validate the certificate. In the second chapter, a paragraph talked about a digital counterpart of credentials in the real world. Such digital credentials must also be protected from being tampered with or faked, credentials of the real world use concepts like holograms for that, but for digital ones, only digital concepts for protecting can be used, also for X.509 certificates. This is what other entries in the X.509 certificate format are for. For instance there is one entry that specifies the time period in which the certificate is valid (certificates have a limited lifetime). So it is easy for other parties to find out if the certificate is still valid or expired. Another important entry is the public key of the party that represents the subject, because X.509 certificates use asymmetric cryptography as another concept for protecting the content. Every

party that uses his own X.509 certificates should have public and private keys for that. Texts that were decrypted with a private key can only be encrypted with the suitable public key and texts that were decrypted with the public key can only be encrypted with the suitable private key. Only the private needs to be protected, the public key can be given to anyone without any risk, that's why it is included in the certificate.

To decrypt a X.509 certificate, a hash value (message digest) of the certificate's content is calculated. This is done by one of many possible suitable algorithms, the name of the used algorithm is also added to the certificate as a special entry, because other parties need this information in order to verify the certificate. These algorithms are sensitive for small changes in the content of a certificate, even if just a single character is changed, the hash-value will differ much from the one of the unchanged version. Although the possibility exists that the hash values of two different texts are equal, this probability is so small that it is ignored. In addition to that, the hash value is also much smaller than the text itself, so it is nearly impossible for attackers to manipulate the certificate or transform the hash-value into the suitable text. The hash-value is then decrypted with the private key of the issuer of the certificate, the result (the digital signature) is also added to the certificate, as well the name of the algorithm used for decrypting.

After receiving, the other party can verify the certificate. To do this, it also calculates the hash-value of the certificate's content. As the algorithm used for that must be the same one that the issuer employed, as described above, a special entry in the certificate contains the name of the algorithm, so the party can determine which is the right one. After calculating the hash value, the digital signature of the certificate is encrypted with the public key of it's issuer, again the same algorithm is used specified in the appropriate entry. If the result of the encryption matches to the calculated hash value, the verification is successful.

With this mechanism, certificates may even be sent over an insecure connection. An attacker can get certificates, but without the appropriate private key, they are useless for him. He may change the certificate's content, but can't calculate the signature. The other party will detect the manipulation by calculating the hash value of the certificate's content and comparing it with the encrypted digital signature of the certificate, as it is described above. And even when not manipulating it, authentication often does not only require a certificate, but also a proof that the party possesses the suitable private key (as done in the TLS handshake for example which is investigated later in this chapter). This is also integrated in the implementation of the authenticatesTo-predicate and will be described in detail later.

Additional entries in X.509 certificates exist, these won't be described, as they don't contribute much value to this Bachelor-thesis, except entries for some extensions, but these will be examined

later as they had to do with another task.

In typical X.509 certificates, the issuer and subject entries are different, not equal (self-signed). This is clear, because in a credential, an authority (organization) (the issuer) states something (abilities, states, ...) about someone (the subject). Comparing to the real world, a driver license card will only be accepted (for instance by the police) if the appropriate organization has issued it. If a driver has created his own card instead and shows it to the police, they won't accept it (if the falsification isn't good enough). As already explained in the second chapter, nearly no one will trust a self-signed credential, as there is no guarantee that people don't lie. This also applies to X.509 certificates, self-signed ones won't be accepted in most authentications, instead certificates are required that are issued by a specific authority that the party trusts. In the internet, special organizations called „Certificate Authorities“ (CAs) exist that are well-known and trusted. They create certificates for customers and enable them to authenticate to other parties that trust the CA. Of course, certificates that are issued by other authorities may be accepted, too. It depends on what a party requires from others (what issuers they accept, ...) in order to authenticate.

To verify a X.509 certificate, the public key of it's issuer is needed. But as already mentioned above, only the public key of the subject is contained in the certificate. When sending a certificate, there is no guarantee that the other party has the public key of the issuer that it needs to verify the certificate. A possible solution for this is to send an additional certificate for the issuer of the first one. This contains the right public key that the other party needs to verify the first certificate. Sending more than one certificate for an authentication is called a certificate chain (or certification path). The major characteristics of such a chain is that the certificate of the sender is at the first position, each following certificate has the issuer of the previous one as subject, so each certificate depends on his predecessor. Instead of restricting a chain to be issued just by one authority, many CAs that depend on each other may appear in this chain, for example CA A has issued a certificate for CA B. As each certificate contains the public key for verifying the previous one, the other party can verify the chain. The only problem might be the last certificate of the chain, if it is self-signed, it can be verified easily, but if not, the other party must possess the suitable public key in order to be able to verify it. In general, both partners in an authentication can profit by such certificate chains, the party who receives it can get the public keys in order to verify the chain. Instead of being restricted to send just one single certificate, the concept of certificate chains gives more flexibility to the user. For instance, when the other party requires a certificate with a special CA as the issuer and the sender doesn't have a certificate with himself as subject and the CA as issuer, he might instead send a suitable certification chain, in which the first certificate's subject is for himself and the last certificate's issuer is the CA.

For example, imagine that *andrew* wants to authenticate to *margret*, to do this, she requires him to

present a certificate (chain) that is issued by the CA e*nTrust*.

andrew has the following certificates:

- A certificate with *andrew* as subject and *trust_agency* as issuer
- A certificate with *trust_agency* as subject and *second_trust_agency* as issuer
- A certificate with *second_trust_agency* as subject and *enTrust* as issuer

andrew has no certificate with himself as subject and *enTrust* as issuer, but he can build a certification chain that satisfies the requirements of *margret*. As the subject of a certificate in a chain must be equal to the issuer of the previous one, *andrew* can't use the first and third certificate to construct the chain. But with the help of second one, he can do it, the requirements described above are fulfilled then. The first certificate has *trust_agency* as issuer which is equal to the subject of the second certificate which issuer *second_trust_agency* matches the subject of the third certificate whose issuer is *enTrust*. Sending the chain with the certificates in this order will satisfy the requirements of *margret* and the authentication will succeed (of course the certificates must be valid, not expired and *andrew* must eventually prove afterwards that he owns the suitable private key for the public key in the first certificate).

X.509 certification chains are supported in many security concepts in the Internet, like SSL or TLS. Also the implementation of the authenticatesTo-predicate supports such chains, but this will be investigated later.

4.1.2 Ideas for integrating the authenticatesTo-predicate

During working on this task (implementation of the authenticatesTo-predicate), two different approaches were developed:

- The first idea was to implement the authenticatesTo-predicate with the help of the TLS/SSL-handshake, because the transmission of a certificate (chain) as well the private key-proof are done automatically in it, these features can be used for the exchange of the certificate (chain) that satisfies the authenticatesTo-predicate.
- The second idea fits better to the existing implementation of and communication in PeerTrust. Instead of using the TLS-handshake (as in the first approach), the transmission and handling of the certificate (chain) is done manually.

Finally, the second approach was chosen for the PeerTrust-prototype and the first one was

skipped. Despite of this, an insight to both solutions will be covered in the following paragraphs because very much time was spent for developing the first approach (more than for the second solution) with the TLS-handshake, so it should also be a part of this Bachelor-thesis.

4.1.2.1 The authenticatesTo-predicate and the TLS-handshake

As already mentioned above, the TLS-handshake [5,6,14] can be used to implement the authenticatesTo-predicate. In the handshake, it is possible for a party to specify which authorities/CAs it trusts. Other parties that want to authenticate must show a certificate (chain) that is signed by those, they also have to prove that they have the right private key for the first certificate in the chain. That handshake is exactly what we need for the implementation of the authenticatesTo-predicate, so this first approach is about integrating this into the PeerTrust-prototype. The TLS-handshake will first be introduced, before focusing on the integration.

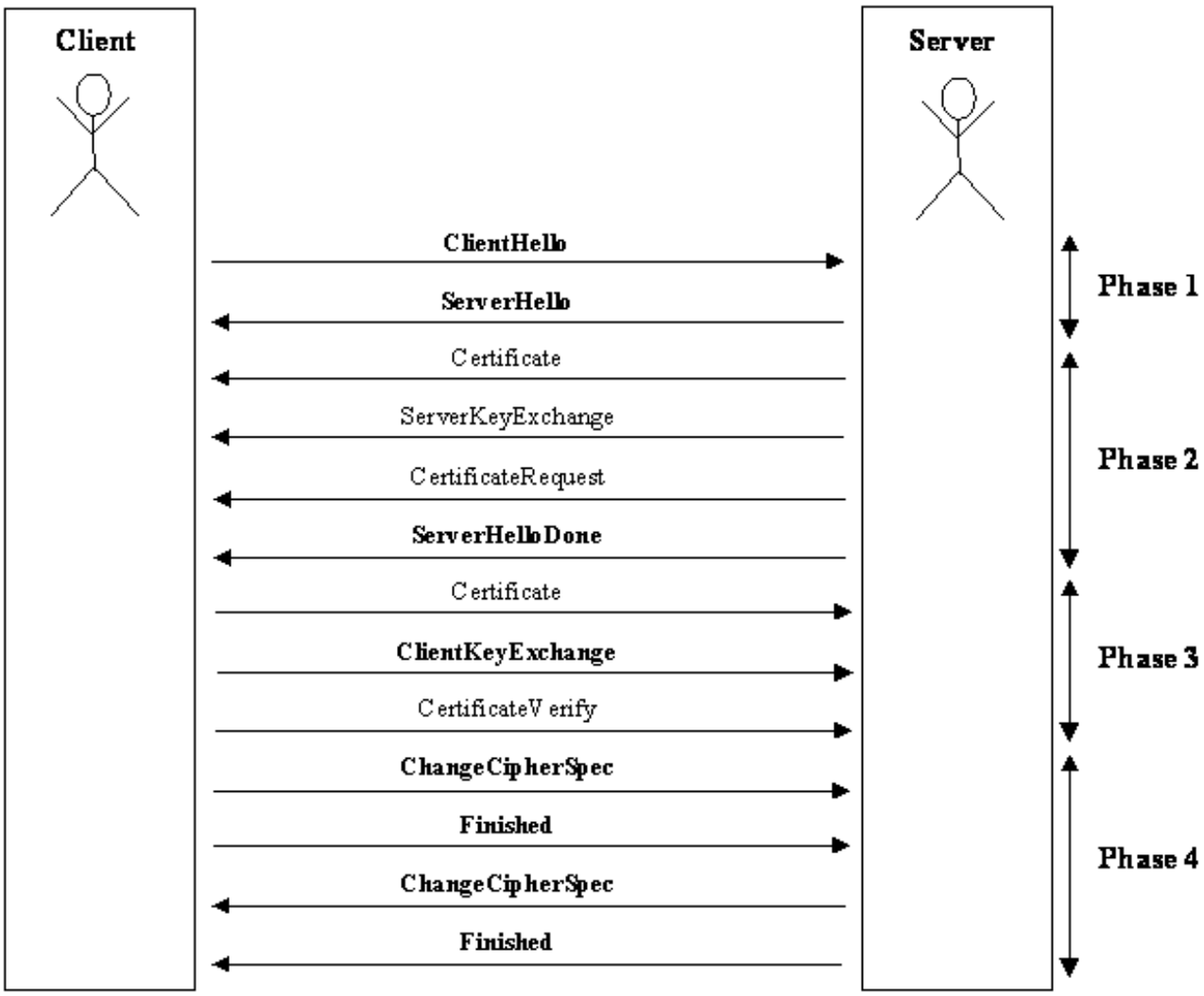
4.1.2.1.1 The TLS-handshake

TLS (Transport Layer Security) [5,6,14] is the successor of the SSL (Secure Socket Layer)-protocol that was invented by Netscape, in fact, it is very similar to SSL V3.0. These protocols help to establish a secure connection between two communicating parties (the data that is exchanged between them is decrypted), online shops for example often use *https* (an extension of http (Hypertext Transfer Protocol) that is based on SSL) to create a secure channel to the customer, no attacker should be able to listen to that connection, as sensitive information like credit card-numbers may be exchanged. TLS offers client/server authentication (this will be investigated in the description about the handshake), confidentiality and data integrity. How exactly the decryption of the data in the secure connection works (this is done with symmetric, secret keys), is beyond the scope of this Bachelor-thesis, instead the handshake and authentication will be focused on.

The TLS-handshake can be seen as the trial of establishment of the secure connection, the authentication of both parties takes place here, as well the negotiation of security parameters that are determined for the encryption and decryption of the transferred data later. Also the exchange of certificates (or certificate chains) is done in this handshake that is the main reason why this introduction to the TLS-handshake was written. If the handshake fails (maybe a party wasn't able to show the wanted certificate (chain) or doesn't support TLS), the secure channel can't be established.

As a lot of handshake variants exist (it would make no sense to describe them all), the following text will focus on the general one operating with RSA-keys (public and private ones), which is also

illustrated in the following picture (required messaged are written **bold**, optional ones are written plain):



Phase 1 represents the start of the handshake that is initiated by the client. Security parameters and requirements from each party are exchanged here.

- The **ClientHello-**message from the client starts the handshake. This message contains the highest SSL/TLS-version the client supports, a session ID (previous sessions may be continued this way; when passing 0 here, a new session is started) and lists of security parameters (for the handshake and the following secure, decrypted channel) that the client supports.
- The **ServerHello**-message is the response from the server, it contains the SSL/TLS-version he has chosen (depending on what information was included in the previous ClientHello-message), the session ID (when the client has passed a 0 before, a new session is started, otherwise the session ID from the client is returned if the server wants to continue the previous session). The

server chooses security parameters from the lists he received from the client.

In **Phase 2**, the server may authenticate himself to the client.

- The optional **Certificate**-message (only needed, when server wants to authenticate himself) contains the certificate (chain) from the server for authentication.
- The optional **ServerKeyExchange** (only needed, when server wants to authenticate himself) is only used when the public keys in the certificates sent before aren't sufficient.
- In the optional **CertificateRequest**-message (only needed, when server wants the client to authenticate himself), the server informs the client that he also requires an authentication from him. He can also specify in this message, which certificate-types and which authorities/CAs he trusts. The client has to present a certificate (chain) that is issued by one of these authorities in the following phase. This is the main reason why TLS is useful for implementing the authenticatesTo-predicate.
- The **ServerHelloDone**-message informs the client that the server-authentication is finished and that it's his turn now.

In **Phase 3**, the client authenticates himself to the server, if he is required to do it.

- In the optional **Certificate**-message (only necessary when the client must authenticate himself to the server), a appropriate certificate (chain) is included, if an invalid one is sent or it isn't issued by one authority that was in the list the server has sent in the CertificateRequest-message, the server cancels the handshake. When the server hasn't sent this message, the client doesn't need to authenticate himself, this is often the case in systems that use TLS, because it's not necessary. But in the implementation of the authenticatesTo-predicate, the server must force the client to authenticate himself.
- In the **ClientKeyExchange**-message, information for creating secret keys for symmetric encryption (this is important for the secure channel that is established after a successful handshake) is sent to the server which is responsible for generating those secret keys (they are used for both encryption and decryption and both parties use the same ones). If RSA-keys are used in the handshake, this information is decrypted with the public key of the server that was received during the previous phase where the server authenticated himself. This can be seen as a test for the server, because he can only encrypt the data if he possesses the corresponding private key.
- The optional **CertificateVerify**-message must only be sent if the client has to authenticate himself. The client must include a proof that he possesses the private key that corresponds to the public key in the first certificate of the chain he sent to the server. He has to create a hash

value for all the messages that were exchanged in this handshake before and decrypt the result with the corresponding private key. After receiving the message, the server calculates the hash value just like the client did and compares it with the encrypted proof from the message. This way, he can determine if he should trust the client (proof was successful) or not.

In **Phase 4**, the client and server exchange security-information for establishing the secure communication channel.

- The client sends the **ChangeCipherSpec**-message to the server in order to inform him that he should change to encryption mode from now on, so that no attacker can understand the further communication.
- The **Finished**-message that the client sends tells the server that he is ready now for establishing the secure communication channel. This message must also contain a hash value of all messages of the handshake exchanged before as an additional test for the client.
- The **ChangeCipherSpec**-message that the server sends informs the client that he should change to encryption-mode from now on.
- The server sends the **Finished**-message to the client to indicate that he establishes now the secure channel. The handshake is finished. From now on, the communication is done over the secure channel.

In general, when a client asks the server to continue a previous TLS-session, the handshake doesn't last as long as a normal one, as not all information has to be generated and exchanged again. It is also possible to perform a handshake over an existing TLS connection, this called a rehandshake then. Either the client or the server can initiate it, a rehandshake may also be ignored, but this is rarely done. There are some reasons for doing a rehandshake, for example creating new symmetric keys to change the encryption for increasing security or the server can also request certificate (chains) from other authorities this way, etc. A client can request a rehandshake by sending a ClientHello-message, the server can initiate it with a ServerHelloRequest-message.

4.1.2.1.2 Using the TLS-handshake to implement the predicate

As it was already explained above what the authenticatesTo-predicate should do and how the TLS-handshake works, the following text will now describe how the first approach (that used TLS) worked in general. Implementation details will be covered in the following chapter. It will also describe why a rehandshake wasn't the best solution unfortunately and a normal handshake had to be used instead.

When a party gets an *authenticatesTo(Identity,Party,Authority)@Requester*-predicate, and is *Party*, it sends a simple message to *Requester* that contains the following information:

- *Identity*
- *Authority*
- The port number to which *Party* must connect to do the TLS-handshake

As a next step, Party (the server) prepares everything for the handshake, the list of authorities (that the server passes to the client during the handshake in the CertificateRequest-message, as already described) is updated, and it now only contains *Authority* as a single entry in order to force the client to show a certificate (chain) that is issued by it. The handshake will occur on the port number that was sent to the client, this port is different from the usual communication.

After receiving the message, *Requester* connects to the specified port and the handshake is performed as it is described above. Of course, the server sends the CertificateRequest-message during the handshake in order to force the client to authenticate himself by showing a certificate (chain) whose issuer is *Authority*. If the client doesn't present a certificate (chain) that fits to these requirements or the handshake fails because of another reason (for instance client or server weren't able to agree on a common SSL/TLS-version in the ServerHello- or Client-Hello-messages, or the client couldn't prove that he has the right private key), the server will consider the authenticatesTo-predicate as being failed. This is also the case if the *Identity*-parameter in the predicate was specified (no variable) and it doesn't match to the subject of the first certificate the client sent to the server during the handshake or the certificate chain isn't valid. Otherwise, the client was able to authenticate successfully and the predicate is fulfilled.

The port and communication channel on which the handshake were made are immediately closed afterwards, because they are only needed for the handshake (and the included automatic exchange of credentials), further communication is done over the facilities PeerTrust provides.

4.1.2.2 Implementing the authenticatesTo-predicate without TLS

The second approach that was developed implements the authenticatesTo-predicate without the help of the TLS-handshake, because the TLS-approach had some technical issues that will be described in the following chapter. This means that the construction and transmission of the client's certificate (chain) had to be done manually. This applies also for the proof the client has to make in order to show that he owns the private key corresponding to the public key of the first

certificate he sent. This second approach has some similarities with the first one. Like in the description of the solution with the TLS-handshake, the following text will only describe how the second approach works in general, implementation details can be found in the next chapter.

When receiving a *authenticatesTo(Identity,Party,Authority)@Requester*-predicate, the party first checks, if it's own identity is *Party*. If not, this query is delegated to it. Otherwise, this party must handle the authenticatesTo-predicate. This is done by sending a special message to *Requester* which contains the following information:

- *Identity*
- *Authority*
- A random text as a challenge for *Requester* to prove that he really owns the private key corresponding to the public key he will send in his certificate (chain) when he tries to authenticate properly later.

When *Requester* receives this special message, he tries to authenticate to the server as specified in the message by building a valid certificate chain. If *Identity* was specified (that means it was instantiated), the subject of the first certificate must be equal to it. The last certificate of the chain must have *Authority* as issuer. Constructing such a certificate chain manually is not trivial, but the exact description of the solution of this Bachelor-thesis is part of the next chapter.

After that, *Requester* tries to prove that he is really the owner of the private key corresponding to the public key that is contained in the first certificate of the chain he just constructed. Therefore, he decrypts the random text that was included in the message he got from *Party* with the suitable private key. As a next step, he sends back an answer to *Party* that includes the following information:

- The subject of the first certificate in the chain
- The issuer of the last certificate in the chain
- The result of the decryption of the random text
- The certificate chain

After receiving the answer, *Party* investigates it. He checks if the first two information of the answer apply to the *Identity*- and *Authority*-parameters of the authenticatesTo-predicate and verifies the certificate chain (if it is not empty, it matches to *Identity* and *Authority* as it is described above, if all certificates chain aren't expired and valid), the verification of the proof tree (second task) is also used for that. Next, the decrypted text from *Requester* is encrypted with the public key

of the first certificate in the chain he sent. If the result is equal to the random text that was sent to *Requester* previously, the proof that *Requester* really owns the corresponding private key is successful. If both the certificate chain and the proof are satisfied as described above, *Party* will consider the authenticatesTo-predicate to be satisfied, otherwise as being failed.

4.2 Verification of the proof tree

The next task in the Bachelor-thesis had to do with the proof tree that is contained in every Tree-object (which corresponds to a query). Such a proof tree contains all necessary information (rules used, instantiations of variables, credentials, ...) to prove that a query or answer was evaluated correct. This is very useful for parties that have sent or delegated a query to another party, as they can verify the answer with the included proof tree. If the answer says that the evaluation was successful, although the proof tree doesn't state this, the party who received the answer will consider it as being failed.

The verification of an answer's proof tree was still missing in the current PeerTrust-prototype, it was one task of this Bachelor-thesis to implement it. Just like in the description of the first task, the following text will only describe the approach in general, because the next chapter covers implementation details.

First, the proof tree-representation in the Tree-class had to be changed because the previous one wasn't sufficient, as necessary credentials and certificates (that belong to signed rules or the authenticatesTo-predicate) couldn't be included in it. The new representation will be covered in the next chapter.

For verifying the proof tree of an answer, the inference engine of PeerTrust is used. Each entry in the proof tree is verified. If it is a signed rule, the corresponding credential will be investigated (if it really fits to the signed rule and if it is valid), checking credentials was also the third task in this Bachelor-thesis, so this solution can be used here. If the verification of the credential fails, the proof tree is not correct. Otherwise, if the current entry is a authenticatesTo-predicate, the corresponding certificate chain will be checked (if the subject of the first certificate fits to the *Identity*-parameter, the issuer of the last certificate is equal to the *Authority*-parameter and if all certificates aren't expired and valid), the verification of the private-key-proof (random text that *Party* has to decrypt with the appropriate private key) isn't done here again. If the verification of the credential chain fails, the proof tree and the corresponding answer will be considered as not being correct. If not, the rule corresponding to the current entry in the proof tree will be added to the knowledge base of the inference engine. If the whole proof tree was investigated and it is still not failed, an existing predicate is executed that checks if the answer can really be inferenced from the

rules that were added just recently to the knowledge base.

4.3 Verification of credentials

The third task in this Bachelor-thesis consisted of verifying credentials, the solution is also used in the previous task because signed rules in a proof tree have a corresponding credential that must also be checked when the proof tree is verified. Again, the following text will not cover the implementation of this solution here, the next chapter is written especially for that.

The task was relatively easy, so the solution is shorter than the ones of the previous tasks, that's because it's description isn't very long. First, it is checked if the credential hasn't expired and is valid. In the current PeerTrust-prototype, credentials are included in X.509 certificates (in the SubjectAlternativeName-extension-entry) and as already mentioned above, checking if a certificate is valid and hasn't expired is easy. To each signed rule belongs also a certificate chain, from which the appropriate public key can be extracted in order to verify the certificate which includes the credential, the certificate chain is verified, too, of course. What is also additionally examined is if the issuer of the credential is equal to the subject of the surrounding certificate.

5. Implementation

The previous chapter focused on how the solutions of this Bachelor-thesis work in general, implementation details were omitted, because they will be investigated this chapter. This is done for each of the following tasks:

- Implementation of the authenticatesTo-predicate
- Verification of the proof tree
- Verification of credentials

The PeerTrust-prototype is implemented in Java, so this language was also used for the solutions of this Bachelor-thesis. The current version of the Java JDK [4] was employed for that, without special IDEs. Publishing the whole source code of these solutions would make no sense here, so parts of it are shown where it's suitable. The javadoc-documentation can be found in Appendix A and on the CD which contains also the PeerTrust-prototype (including the solutions of this Bachelor-thesis) with source code.

5.1 Implementation of the authenticatesTo-predicate

Two approaches (that were covered in the previous chapter) were developed for implementing the authenticatesTo-predicate:

- The approach with the TLS-handshake
- The manual approach

The second one was preferred for the PeerTrust-prototype (reasons for this will be covered later) and the first one was skipped. Implementation details for both solutions will be investigated here, even if the first solution isn't used any more, the work on it took very much time (more than for the second one), and so it should also be a part of this Bachelor-thesis.

5.1.1 The approach with the TLS-handshake

Using a TLS-handshake for implementing the *authenticatesTo(Identity,Party,Authority) @Requester*-predicate seem to be an ideal solution, because the server has the ability to specify which authorities/CAs he trusts (in the CertificateRequest-message), the client has to present a certificate (chain) then that is issued by one of these authorities and he has to prove that he possesses the right private key. That is exactly what we need for the implementation of the

predicate, as *Requester* has to present a chain that must be issued by *Authority*. As all these functionalities are already included in the TLS-handshake, the specification for CAs, the construction and transmission of an appropriate certificate (chain), as well the private-key-proof mustn't be reimplemented.

As the TLS-handshake-protocol is very complex [6,14], reimplementing it would have taken too much time, so an existing TLS-extension for Java was searched for. The only suitable candidates which could be found were PureTLS [15] and JSSE [4,5]. The latter one was chosen, because PureTLS needs external programs to run (which is bad, as PeerTrust is a signed applet), certificate checking is primitive and PureTLS a library from a third party which hasn't been updated for a long time. In contrast to that, JSSE (Java Secure Socket Connection) is included in the Java JDK since the version 1.4.0 (it was an external package before), doesn't have the disadvantages of PureTLS and the documentation is much better, also the amount and functionality of the classes.

In JSSE, special factory classes exist for generating objects of certain classes, as new providers, algorithms or SSL/TLS-versions may be added to JSSE. Although also support for https exists, the first authenticatesTo-predicate-solution will focus on using TLS in socket communication. Beside many classes that allow access to parameters, information of SSL/TLS and the handshake, special extended classes exist that make the normal socket communication in Java secure (by using SSL/TLS and the corresponding handshake). So these special sockets are used in the solution, also the parameters for the handshake have to be manipulated (because the list of authorities/CAs that the server accepts must be changed).

When an authenticatesTo-predicate must be evaluated, *Party* is responsible for forcing *Requester* to authenticate himself and handle his answer. Therefore, *Party* sends a special message to *Requester* with all the information it needs to authenticate (*Identity*, *Authority* and port number). The server already uses a serversocket that watches incoming queries and answers from other parties. We can't use this one for the handshake with *Requester*, because the manipulation of the authority list (that is necessary for forcing *Requester* to show a certificate (chain) issued by *Authority*) would also affect queries and answers from other parties then (they would also have to show a certificate (chain) issued by *Authority* then). So the handshake with *Requester* will occur on another serversocket and port then. Creating a SSL/TLS-serversocket (*SSLServerSocket*-class in JSSE) is done in several steps, here the classes from JSSE [4,5] (most can be found in the javax.net.ssl-package) come into play. The exact serversocket-creation will be described backwards (as already explained above, much factory classes exist):

- A *SSLServerSocket* can be obtained from a *SSLServerSocketFactory*

- A *SSLServerSocketFactory* can be obtained from a *SSLContext*
- A *SSLContext* must be initialized by an array with objects of the class *KeyManager* and an array with *TrustManager*-objects
- A *KeyManager* is responsible for deciding which public key and corresponding certificate (chain) should be shown to other parties in the handshake. A *KeyManager* can be obtained from a *KeyManagerFactory* or a custom one can be used by implementing the *KeyManager*- or *X509KeyManager*-interface
- A *TrustManager* is responsible for deciding whether to trust the certificate (chain) from the other party in the handshake or not. A *TrustManager* can be obtained from a *TrustManagerFactory* or a custom one can be used by implementing the *TrustManager*- or *X509TrustManager*-interface
- *TrustManager* and *KeyManager* need a *KeyStore* (in the java.security-package) which contains local private keys, local certificates or certificates from other parties

The standard *KeyManager* that can be obtained from the corresponding factory class is sufficient, but as the list of authorities/CAs the server trusts in the handshake must be modified, a custom *TrustManager* had to be written by creating a class MyTrustManager that implements the *X509TrustManager*-interface (because PeerTrust uses X.509 certificates). This class checks if the certificate (chain) from *Requester* was really issued by *Authority* and if the first certificate's subject is *Identity*, if specified. After having created the *SSLServerSocket* with the help of the custom *TrustManager*, a special method of the *SSLServerSocket* forces the server to send the CertificateRequest-message in the handshake in order to tell the client that he must authenticate himself. The *SSLServerSocket* now waits until *Requester* connects to it, the *SSLSocket* will do the handshake then. The creation of the *SSLServerSocket* and the handling of the handshake was implemented as a function (that *Party* uses) that returns if the handshake (and so the authenticatesTo-predicate) has succeeded or not. After the handshake is performed, the socket connection is immediately closed, because only the result of the corresponding handshake is needed. The source code of this function:

```
/**
 * Method for the server that prepares everything for the handshake and returns
 * if it has succeeded or not.
 * @param int The number of the port that should be used for the serversocket.
 * @param Identity The identity and authority (attributes of class Identity) that the
 *        client must satisfy.
 * @return boolean If the handshake succeeded or not.
*/
public boolean initNewHandshake(int port,Identity identity) {
```

```
        try {
                //Load the keystore
                KeyStore ks=KeyStore.getInstance("JKS");
                ks.load(new FileInputStream(strKeyStoreFile),
                        strStorePassword.toCharArray());
                //Use a factory to obtain a default KeyManager
                KeyManagerFactory kmf=KeyManagerFactory.getInstance(
                        SecureServerSocket.ALGORITHM);
                kmf.init(ks,strKeyPassword.toCharArray());
                //Create and initialize a SSLContext with a custom TrustManager and
                //default KeyManager
                SSLContext sslcontext=SSLContext.getInstance("TLS");
                sslcontext.init(kmf.getKeyManagers(),new TrustManager[]{
                        new MyTrustManager(ks,identity)},null);
                //Create the serversocket
                SSLServerSocket serversocketTemp=(SSLServerSocket)sslcontext.
                        getServerSocketFactory().createServerSocket(port);
                serversocketTemp.setNeedClientAuth(true);
                SSLSocket socketTemp=(SSLSocket)serversocketTemp.accept();
                //If the client has connected, start the handshake
                socketTemp.startHandshake();
                socketTemp.getSession().invalidate();
                socketTemp.close();
                serversocketTemp.close();
        }
        catch(Exception e) {
                System.err.println("AuthorityRequestServer: rehandshake failed: "+e);
                return false;
        }
        return true;
    }
```

Implementing the handling of the authenticatesTo-predicate for *Requester* was easier. The communication between PeerTrust-parties is done by existing, custom classes named *SecureClientSocket* and *SecureServerSocket* that also use these secure Java-sockets (*SSLSocket*). So, after receiving the special message from *Party* that tells *Requester* to show a certificate (chain) that satisfies the authenticatesTo-predicate, the *SecureClientSocket* can be used to connect to the address and port of *Party* (these information were included in the special

message). As *Party* has prepared everything for the handshake and is responsible for handling it (and the authenticatesTo-predicate), *Requester* must only connect to him, so that the handshake can start. The exchange of the suitable certificate (chain) that Party wants to see is done automatically in the handshake thanks to JSSE.

A handshake had to be used instead of a rehandshake because the first contact between *Requester* and *Party* is done when *Receiver* receives the special message (which includes *Identity*, *Authority* and port to connect to) from *Party*. *Requester* is the server (for incoming queries and answers exists a serversocket, as already mentioned above) then and *Party* the client (as he has connected to this serversocket). Because only the server can specify in a handshake which authorities/CAs he trusts, roles (server and client) have to be switched here (this is impossible for *SSLSocket*-objects in Java after the first handshake): *Party* must be the server. That's why a new connection has to be established between them (with roles switched), a rehandshake can only be done over an existing connection, so a normal handshake has to be used instead.

Depending on if the handshake succeeded or not, *Party* will consider the authenticatesTo-predicate as being satisfied or failed. If the query was delegated to *Party* from another party, the answer will be sent back.

5.1.2 The manual approach

Unfortunately, the first approach with the TLS-handshake has some disadvantages. Using only the existing communication system for the handshake wasn't possible, it had to be extended (as described above) in order to make it work. The current PeerTrust-prototype uses secure sockets (*SSLSocket*) for communication (what is perfect for the first approach), but in the future, this may change, so relying on the use of secure sockets doesn't seem to be a good idea. Instead, the implementation should work with the existing communication system independent from its implementation. That is exactly what the second approach does. Another problem of the solution with the TLS-handshake is the use of the second server socket that is needed for the handshake. It increases complexity, memory usage and processing power (imagine several handshakes with different parties have to be processed at the same time). Again, a solution that uses the existing communication system of PeerTrust without relying on a specific implementation should be preferred, because it doesn't waste so many resources than the TLS-approach.

The second approach for implementing the *authenticatesTo(Identity,Party,Authority)@Requester*-predicate fulfills the requirements of the previous paragraph, it just uses the existing communication system of PeerTrust without extending it, relying on the use of secure sockets and

wasting too much resources. This leads to the conclusion that the TLS-handshake can't be used any more, as it needs secure sockets or https in Java (specific implementations for which the communication in PeerTrust must be extended as described above). We used the handshake, because the construction and transmission of the certificate (chain) that satisfies the authenticatesTo-predicate was done automatically, also the proof of *Requester* that he really owns the suitable private key for the public key that is contained in the first certificate sent to *Party*. These concepts must be implemented manually in the second approach that doesn't use the handshake for these tasks any more.

When a party receives an authenticatesTo-predicate, it delegates the query (to *Party*) if the local peer isn't the one specified in the *Party*-parameter of the predicate. Otherwise, it is responsible for evaluating this predicate. To do this, it sends a special message to *Requester* in order to inform him that he has to show a suitable certificate (chain) to satisfy the authenticatesTo-predicate. This special message contains the following information that *Requester* needs to authenticate himself:

- *Identity*
- *Authority*
- A random text

This random text represents a challenge for *Requester* for proving that he has the suitable private key that corresponds to the public key in the first certificate he sends. How this proof works will be described later. The source code for creating the random text:

```
Random random=new Random();
byte enc_bytes[]=new byte[Math.abs(random.nextInt())%31+20];
random.nextBytes(enc_bytes);
String enc_str=new String(enc_bytes);
```

The Tree-object (that belongs to the authenticatesTo-query *Party* has to evaluate) stores the predicate and the message sent to *Requester* in a special lastExpandedGoal-attribute (which stores queries and delegations sent to other peers). When *Party* receives the answer from *Requester* later, he must verify it, therefore he needs the predicate and the special message sent, this will be described later in more detail. The status of the Tree-object is also set to waiting-mode, as this object waits for an answer and can only be further processed locally after obtaining it.

After receiving the special message from *Party*, *Requester* now knows that he has to authenticate himself according to the information that were contained in the message. As a first step,

Requester tries to build a certificate chain whose last certificate is issued by *Authority* and whose first certificate's subject is *Identity* (if it is instantiated and no variable; as Prolog is used, variables start with an upper case character). Building a certificate chain is not trivial. A special class for certificate chains (named CertificateChain) was created during the work on this Bachelor-thesis that automatically constructs a chain and stores it. Depending on if an authority was specified or not, the way the chain is built differs. The first case will be described here, the second one (if the authority-parameter is null, not specified) is part of the second task of this Bachelor-thesis and will be covered later. Here is the pseudo code of the first case (it's easier to understand than the Java code for it):

```
//-identity is the subject that the first certificate of the chain must have, if
//        subject is null, the subject can be anything
//-authority is the issuer that the last certificate of the chain must have
//-keystore is the keystore which contains all local certificates and private keys
//-vectorCerts represents an out-parameter, the certificate chain will be stored
//        there after the method has finished. This Vector will then be transformed
//        in an array afterwards that is an attribute of the class and represents the
//        constructed certificate chain.
void searchCompleyChain(String identity,String authority,KeyStore keystore,
        Vector vectorCerts) {
        Vector vectorTree=new Vector();
        Vector vectorTemp=new Vector();
        clear vectorCerts
        //All entries with a private key in the keystore are examined, as these
        //are potential starting points to build the chain
        iterate through all entries in the keystore {
                if current entry is not an entry for a private key
                        continue;
                //An entry with a private key may have a certificate chain that might
                //be the one we search for, this is checked here
                if current entry is a certificate chain, its length is greater than 0, it
                        consists of X.509 certificates, the first one is valid and (identity
                        is null or the first certificate's subject is identity) {
                        iterate through the certificates in the certificate chain {
                                if current certificate is no X.509 certificate or certificate is
                                        not valid
                                        break;
                                if the issuer of the current certificate is authority {
```

```
                        add all previous certificates from the chain and
                            the current certificate to vectorCerts
                        return;
                    }
                }
            }
            //An entry with a private key may have a certificate that may be a
            //valid chain or can be a starting point for one
            if current entry is a X.509 certificate {
                if certificate is valid and (identity is null or the certificate's
                    subject matches identity) {
                    if the certificate's issuer is authority {
                        add certificate to vectorCerts
                        return;
                    }
                    else
                        add certificate to vectorTemp
                }
            }
        }
        //If a valid certificate chain was found in the entries of the keystore that
        //corresponds to a private key, the method would have returned before.
        //Otherwise, certificates that are potential candidates for being starting
        //points of a chain are stored in the vectorTemp-Vector. When no such
        //candidates were found, the chain can't be built.
        if vectorTemp is empty
            return;
        add vectorTemp to vectorTree
        //The certificate chain is built here
        while(true) {
            //When there are no more possibilities to search for a valid
            //certificate chain, quit
            if vectorTree is empty
                return;
            //Get current candidate list
            vectorTemp=last element of vectorTree-Vector
            //If candidate list is empty, delete it from vectorTree
            if vectorTemp is empty {
```

```
                    remove last element from vectorTree-Vector
                    remove last element from vectorCerts-Vector
                    continue;
                }
                String temp_issuer=issuer from first certificate in vectorTemp
                add first certificate in vectorTemp to vectorCerts
                remove first certificate from vectorTemp
                //If temp_issuer is authority, we have found a certificate chain
                if temp_issuer equals authority
                    return;
                //Create a new list with all new candidates found in the keystore. The
                //candidates must be valid successors of the current certificate.
                vectorTemp=new Vector();
                iterate through all entries in the keystore {
                    if current entry is a X.509 certificate, it is valid, and it's
                        subject is equal to temp_issuer
                        add certificate to vectorTemp
                }
                add vectorTemp to vectorTree
            }
        }
```

This method isn't trivial, so an example will be presented here in order to make it more understandable. Imagine, a party named steve has the following certificates in his keystore:

- Certificate A: subject: steve, issuer: mary (+ private key)
- Certificate B: subject: steve, issuer: marc (+ private key)
- Certificate C: subject: mary, issuer: john
- Certificate D: subject: marc, issuer: amanda

Certificates A and B have corresponding private keys in the keystore. Imagine steve must build a certificate chain, **identity** is **null** (doesn't matter) and **authority** is **amanda**. The algorithm first examines certificates in the keystore that correspond to private keys in order to find potential starting points for constructing the chain. As the identity is specified as null, a certificate's subject must not match to it. So after this examination, the vectorTemp-Vector looks like this, because only certificate A and B satisfy the criteria described above (certificate corresponds to private key):

vectorTemp: [A, B]

This Vector can be seen as a candidate list, all entries (certificates) in it will be examined. vector Temp is then added to vectorTree:

vectorTree: [[A,B]]

This Vector stores candidate lists, so that all possibilities for a certificate chain can be examined (for each candidate in a candidate list, a new candidate list is calculated). Now the method enters the while(true)-loop. The last candidate list in vectorTree ([A, B]) is examined. The issuer of the first certificate of it is stored (in this case it's mary, issuer of certificate A), then the certificate is added to vectorCerts (which contains the valid chain if the method finishes) and deleted from the candidate list, that also affects vectorTree.

vectorCerts: [A]
vectorTree: [[B]]

The issuer (mary) that was stored above doesn't match amanda (authority), so the keystore is examined to create a new candidate list with certificates whose subject is mary. Certificate C fulfills this requirement, it's subject is certificate A's issuer, so A and C represent a certificate chain. The new candidate list (which includes only certificate C) is added to vectorTree, after examining the keystore:

vectorTree: [[B], [C]]

In the new iteration of the while(true)-loop, the last candidate list of vectorTree ([C]) is examined. The issuer of the first certificate (john) in it is stored again, it isn't equal to amanda, so the chain isn't valid yet. Certificate C is added to vectorCerts and deleted from the candidate list:

vectorCerts: [A, C]
vectorTree: [[B], []]

The keystore is examined for new candidates for certificate C (the candidate's subject must be john), but no candidates can be found and the (empty) candidate-list is added to vectorTree:

vectorTree: [[B], [], []]

In the new iteration of the while-loop, the last candidate list of vectorTree is empty, so this candidate list as well the last entry in vectorCerts is deleted:

vectorCerts: [A]
vectorTree: [[B], []]

In the new iteration of the while-loop, the last candidate list of vectorTree is empty, so this candidate list as well the last entry in vectorCerts is deleted:

vectorCerts: []
vectorTree: [[B]]

The certificate chain A and C didn't work, in the next iteration certificate B is examined as it is the only element in the last candidate list of vectorTree. It's issuer (marc) isn't equal to authority (amanda), vectorCerts and vectorTree are updated as follows:

vectorCerts: [B]
vectorTree: [[]]

After examining the keystore a new candidate list (certificate D) for B is found:

vectorTree: [[], [D]]

In the next iteration of the while-loop, a suitable certificate chain is found, because certificate D is examined next (as it is first entry in the last candidate list of vectorTree) and it's issuer is amanda (authority). Before quitting the method, vectorCerts (which contains the suitable certificate chain afterwards) and vectorTree are updated:

vectorCerts: [B, D]
vectorTree: [[], []]

Coming back on how *Receiver* handles the authenticatesTo-predicate, after getting the special message from *Party* and building the certificate chain with the help of the class and method just described, *Receiver* must find the private key that corresponds to the public key of the first certificate in the chain. The CertificateChain-class could include the right private key as an attribute, but this would be a bad idea because an object of this class will be sent to *Party* who shouldn't get this key. So *Requester* examines the keystore again and stores all private keys in a Vector. This key is needed to decrypt the random text that was contained in the special message in order to prove to *Party* that *Requester* has the suitable private key for the first certificate in the chain. So *Receiver* decrypts the random text (the Signature-class in the java.security-package is

used for that) with all the private keys he has, until he has found the one that corresponds to the public key. The code for decrypting:

```
Signature signature=Signature.getInstance(CRYPT_ALGORITHM);
for(int i=0;i<vectorPrivateKeys.size();i++) {
        //decrypt data with private key
        signature.initSign((PrivateKey)vectorPrivateKeys.elementAt(i));
        signature.update(enc_str.getBytes());
        byte dec[]=signature.sign();
        //Verify the decrypted data with the public key
        signature.initVerify(publickey);
        signature.update(enc_str.getBytes());
        if(signature.verify(dec)) {
                ...
                break;
        }
}
```

After that, *Requester* sends back the answer to *Party* as an predicate which has the following information as parameters:

- The subject of the first certificate in the certificate chain
- The issuer of the last certificate in the certificate chain
- The decryption of the random text

The certificate chain (included in a special class that will be covered later, in the description of the proof tree-verification-implementation) is attached to the answer's proof tree.

This answer from *Party* enables *Requester* to verify the information and to decide if it satisfies the authenticatesTo-predicate. He compares the special message he sent before to this answer. If the answer specifies another authority or identity (except it was a variable in the message), the predicate fails. Next, the decrypted text from *Party* is encrypted with the public key of the first certificate in the certificate chain (which is included in the AuthenticatesToProofRule-class which will be explained later). If the encryption and the random text that was sent in the special message before don't match, the proof from *Requester* fails and so does the authenticatesTo-predicate. The code for verifying the proof:

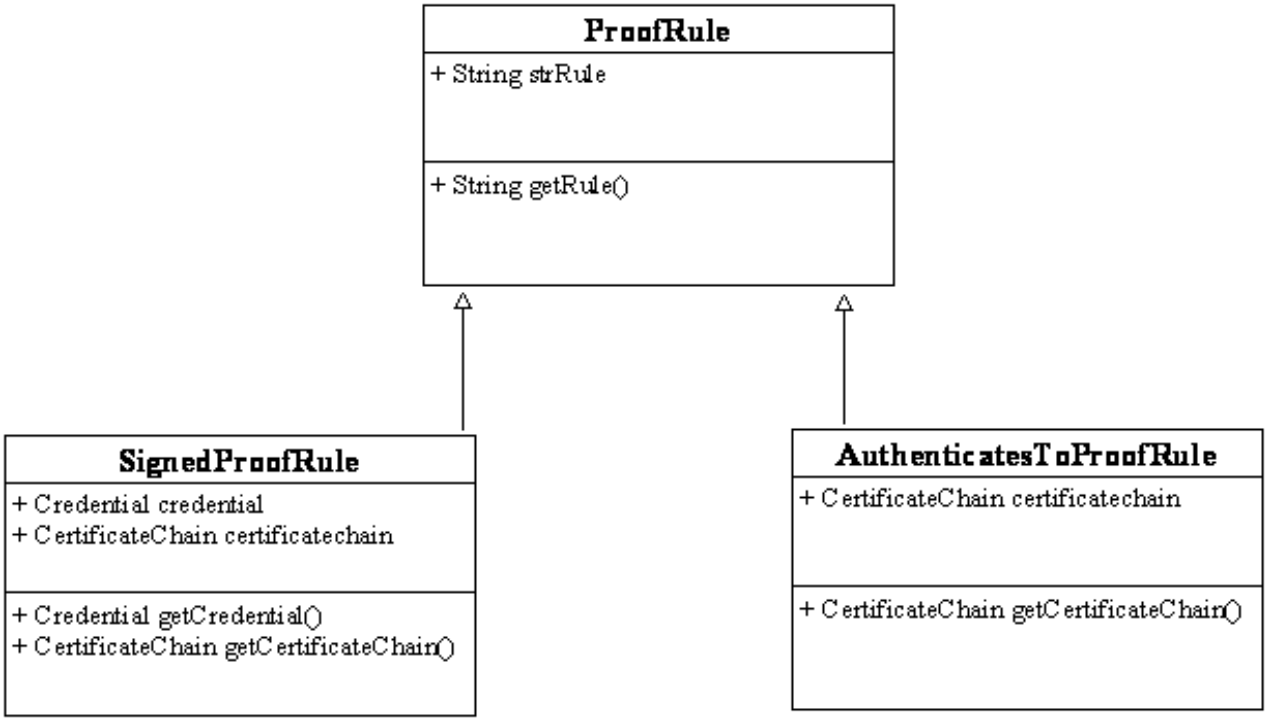
```
//Get the right certificate chain from the proof tree
```

```
vector=answer.getProofRuleVector();
if((vector.size()==0)||!(vector.elementAt(vector.size()-1) instanceof
        AuthenticatesToProofRule))
        return false;
AuthenticatesToProofRule authrule=(AuthenticatesToProofRule)
        vector.elementAt(vector.size()-1);
X509Certificate certs[]=authrule.getCertificateChain().getCertificates();
try {
        ...
        //Check the decrypted string from Requester
        Signature signature=Signature.getInstance(CRYPT_ALGORITHM);
        signature.initVerify(certs[0].getPublicKey());
        //Compare the original text
        signature.update(enc_str.getBytes());
        //to the encyption of the decrypted text from Requester
        signature.verify(dec);
}
catch(Exception e) {
        e.printStackTrace();
        return false;
}
```

The checking of the certificate chain (if *Identity* matches to the subject of the first certificate, if *Authority* is equal to the issuer of the last one and if the chain is valid) is done in the verification of the answer's proof tree (that's what the second task in this Bachelor-thesis was about), so it will be covered in the corresponding text below.

5.2 Verification of the proof tree

The second task had to do with the verification of proof trees. A party can check if the answer to a previously sent query is correct by examining the included proof tree. First, the proof tree representation in the Tree- and Answer-classes had to be changed, as the existing implementation (proof tree is just a single String) was sufficient for local rules, but not for signed ones (as they include a credential and a certificate chain) and authenticatesTo-rules (because of the certificate chain). The new representation consists of a Vector that stores all rules (normal and signed ones, the authenticatesTo-rules) and the certificate chains and credentials that may be associated with them. Special classes were created for that, which can be seen in the following illustration:



The ProofRule-class is created for storing the local rule as a String. As the other two classes extend ProofRule, they also have a text representation of rules. The authenticatesTo-predicate-implementation needs a certificate chain sent from *Requester* to *Party* (as already described), this chain is sent in the proof tree of the answer, so the AuthenticatesToProofRule-class had to be created that includes the certificate chain (the same class (CertficateChain) is used for this chain that was described in the section of the authenticatesTo-predicate-implementation). The SignedProofRule-class (for signed rules) has the corresponding credential and also a certificate chain. A special method for the Tree-class was written during the work on this Bachelor-thesis that allows to easily add new entries to the proof tree. A single rule or multiple ones can be passed as a parameter, the method then transforms them into ProofRule- or SignedProofRule-objects (AuthenticatesToProofRule-objects are not supported by this method, they must be directly added to the Vector) and searches for the right credentials and certificate chains, if appropriate. Here is the source code:

```
/**
 * New rules can be added to the proof tree with this method. AuthenticatesToProofRule-
 * objects can't be created with this method, so the authenticatesTo-predicate
 * shouldn't be used as a parameter.
 * @param CredentialStore The place where the credentials are stored.
 * @param Configurator A configuration-object.
 * @String The (normal or signed) rule in text that should be added. These may be
```

```
 *       also be more than one, in that case, the rules must be separated by
 *       commas. All rules must be enclosed by brackets, for example:
 *              [rule1(...),rule2(...),...]
 */
public void appendProofRuleVector(CredentialStore credentialstore,
        Configurator config,String stringProof) {
        if(vectorProofRules==null)
                vectorProofRules=new Vector();
        int offsetstart=1,num_brackets=0,index;
        char c;
        String str,str2;
        //Try to extract all rules that are contained in the parameter
        for(int i=1;i<stringProof.length()-1;i++) {
                c=stringProof.charAt(i);
                if((c=='(')||(c=='['))
                        num_brackets++;
                else if((c==')')||(c==']'))
                        num_brackets--;
                //New rule is parsed
                if(((c==',')||(i==stringProof.length()-2))&&
                        (i>offsetstart)&&(num_brackets==0)) {
                        str=stringProof.substring(offsetstart,(c==',') ? i : i+1);
                        str=str.replaceAll(" ","");
                        index=0;
                        //All hidden-terms have to be replaced by []
                        while((index=str.indexOf("hidden",index))!=-1)
                                if(((str.charAt(index-1)=='(')||(str.charAt(index-1)
                                        ==','))&&((str.charAt(index+6)==',')||
                                        (str.charAt(index+6)==')')))
                                        str=str.substring(0,index)+"[]"+str.substring(
                                        index+6);
                        //If rule is a signed one
                        if(str.startsWith("signed(")) {
                                str2=str.replaceAll("\\(r\\(","(rule(");
                                //Try to find the corresponding credential
                                Credential credential=credentialstore.getCredential(
                                        str2);
                                if((credential==null)&&((index=str2.lastIndexOf("@"))
```

```
                                        !=-1))
                                        credential=credentialstore.getCredential(
                                                str2.substring(0,index));
                                //Add special object to the proof tree
                                vectorProofRules.addElement(new SignedProofRule(str,
                                        credential,config));
                        }
                        else
                                //Add special object to the proof tree
                                vectorProofRules.addElement(new ProofRule(str));
                        offsetstart=i+1;
                }
        }
}
```

In contrast to the construction of a chain for the authenticatesTo-predicate, the algorithm for building one for a signed rule is much simpler, because no special authority must be specified that issued the chain. The main concept of this algorithm will be described instead of showing source code: If a suitable credential chain for the credential already exists in the keystore, this one will be used, otherwise, the chain has to be built manually. Therefore, the issuer of the certificate which contains the credential is stored. Next, the keystore is examined for a certificate that has this issuer-value as a subject, then it's issuer is stored, and so on. This is done until no suitable next certificate can be found. This behavior finds just one possible chain (probably not the best one) for the credential, but this is sufficient, as the chain is used only for getting the public key to verify the certificate with the credential.

The verification of the proof tree is done in the following method, it is contained in a class named ProofTreeValidator whose purpose is verifying a proof tree:

```
/**
 * Checks, if a Tree is correct by verifying its proof tree.
 * @param Tree The tree whose proof tree should be verified.
 * @param Configurator The configuration-object of the local peer.
 * @return boolean If the Tree has a valid proof tree.
 */
public static boolean isProofTreeOk(Tree tree,Configurator config) {
        ProofRule proofrule;
        try {
```

```
                //Create and initialize the inference engine
                MinervaProlog engine=new MinervaProlog(config);
                engine.loadFile("proof");
                //Get the proof tree of the Tree-object and iterate through its entries
                Vector vector=tree.getProofRuleVector();
                for(int i=0;i<vector.size();i++) {
                        proofrule=(ProofRule)vector.elementAt(i);
                        //If current rule is a signed one and is not correct, quit
                        if((proofrule instanceof SignedProofRule)&&
                                (!checkSignedProofRule((SignedProofRule)proofrule)))
                                return false;
                        //If current rule is an authenticatesTo one and is not correct,
                        //quit
                        else if((proofrule instanceof AuthenticatesToProofRule)&&
                                (!checkAuthenticatesToProofRule(
                                (AuthenticatesToProofRule)proofrule,config)))
                                return false;
                        //Add rule to knowledge base of the inference engine
                        engine.execute("asserta("+proofrule.getRule()+")");
                }
                //Check if last expnded goal of the rule can be inferenced from the rules
                //just added to the inference engine
                return engine.execute("proof("+tree.getLastExpandedGoal()+","+
                        tree.getRequester().getAlias()+")");
        }
        catch(Exception e) {
                e.printStackTrace();
        }
        return false;
}
```

After initializing the Minerva Prolog inference engine, the method iterates through the entries in the proof tree-Vector. If the current object is from the class SignedProofRule or AuthenticatesToProofRule, special methods (that will be described below) check it and may cause the verification of the proof tree to fail. The text representation of each rule is added to the knowledge base of the Minerva Prolog-engine. After all entries have been investigated and were ok, an existing predicate will be called that checks if the last expanded goal of the tree can really be inferenced from the entries in the knowledge base.

A special method was implemented for checking the AuthenticatesToProofRule (as indicated above), the part in a proof tree that corresponds to an authenticatesTo-predicate. As mentioned in the section of the authenticatesTo-predicate-implementation, this method is used for verifying the certificate chain that *Requester* sent to *Party*. This method is also called from the one described above. Here is the source code:

```
private static boolean checkAuthenticatesToProofRule(AuthenticatesToProofRule
        authrule) {
        //Get the certificate chain and quit if it is empty
        X509Certificate certs[]=authrule.getCertificateChain().getCertificates();
        if(certs.length==0)
                return false;
        //Parse identity and authority out of the parameters of the rule
        String str=authrule.getRule();
        int a=str.indexOf("("),b=str.indexOf(",",a);
        String identity=(a!=-1)&&(b!=-1)&&(b-a-1>0) ? str.substring(a+1,b) : "";
        a=str.indexOf(",",b+1);
        b=str.indexOf(")",a);
        String authority=(b!=-1)&&(b-a>0) ? str.substring(a+1,b) : "";
        //If the first certificate's subject isn't identity and the last one's issuer isn't
        //authority, quit
        if((!CertificateChain.getSubjectAlias(certs[0]).equals(identity))||
                (!CertificateChain.getIssuerAlias(certs[certs.length-1]).equals(
                authority)))
                return false;
        try {
                for(int i=0;i<certs.length;i++) {
                        //Check if certificate isn't expired
                        certs[i].checkValidity();
                        //Each certificate must be verified with the public key of
                        //its successor
                        if(i<certs.length-1)
                                certs[i].verify(certs[i+1].getPublicKey());
                        //The last certificate chain must be verified with the public key of
                        //the issuer which must be in the keystore
                        else
                                return checkLastCertificate(certs[i],config);
```

```
                }
            }
            catch(Exception e) {
            }
            return false;
    }
```

This method checks if the first certificate's subject is *Identity* and the issuer of the last certificate of the chain is *Authority*, a requirement for satisfying the authenticatesTo-predicate. Next, the certificate chain is investigated if it is really correct by examining if all certificates aren't expired. Each certificate is validated with the public key of the next one, except the last one, *Party* searches for the suitable public key in his keystore in order to verify it. Attackers may fake the last certificate by creating one who has the issuer *Authority*, but his own private key has signed it, this behavior is not possible when *Party* is responsible for finding the suitable public key to verify the last certificate of a chain. The checkLastCertificate-method implements this behavior, it tries to find the public key (for the issuer of the last certificate) in the keystore and cerifies the last certificate with it.

5.3 Verification of credentials

The last task in this Bachelor-thesis was to verify credentials. As those are contained in X.509 certificates and are included in a proof tree as an attribute of a SignedProofRule-object (the class is introduced above), this task and the previous one overlap. In the algorithm that verifies the proof tree, a method is called that checks SignedProofRule-objects in the tree. In fact, this method belongs to this task, so it will be described here. First, the source code:

```
    private static boolean checkSignedProofRule(SignedProofRule signedproofrule) {
            //Get the credential, if it doesn't exists, quit
            Credential cred=signedproofrule.getCredential();
            if(cred==null)
                    return false;
            //Check if the certificate that contains the credential is valid
            X509Certificate cert;
            try {
                    cert=(X509Certificate)cred.getEncoded();
                    cert.checkValidity();
            }
            catch(Exception e) {
```

```
        e.printStackTrace();
        return false;
}
String subject=CertificateChain.getSubjectAlias(cert),
        issuer=CertificateChain.getIssuerAlias(cert);
String rule=signedproofrule.getRule();
rule.replaceAll("\\(r\\(","(rule(");
//Check if rule is really the one specified in the credential
if(!cred.getStringRepresentation().equals(rule)) {
        int a=rule.lastIndexOf("@");
        if((a==-1)||(cred.getStringRepresentation().equals(
                rule.substring(0,a))))
                return false;
}
//Check if the rule is really issued by the issuer of the certificate
if(cred.getStringRepresentation().indexOf("@"+subject+",")==-1)
        return false;
//Get the certificate chain, if it doesn't exist, quit
CertificateChain certchain=signedproofrule.getCertificateChain();
if(certchain==null)
        return false;
X509Certificate certs[]=certchain.getCertificates();
//The certificate which includes the credential must either be verified
//successfull with it's own public key (if it's a self-signed one) or with the
//public key of the next certificate in the chain
try {
        cert.verify(certs[0].getPublicKey());
}
catch(Exception e) {
        try {
                cert.verify(certs[1].getPublicKey());
        }
        catch(Exception exc) {
                return false;
        }
}
//Verify alll the certificates in the chain, if they aren't expired and can be
//verified with the public key of the successor
```

```
        try {
                for(int i=0;i<certs.length;i++) {
                        certs[i].checkValidity();
                        if(i<certs.length-1)
                                certs[i].verify(certs[i+1].getPublicKey());
                }
        }
        catch(Exception e) {
                return false;
        }
        return true;
}
```

This method checks if the certificate that contains the credential isn't expired, if the text presentation of the rule is equal to the content of the credential and if the issuers of the rule and the certificate match. The certificate chain is used for verifying the certificate which contains the credential, also the chain itself is examined.

6. Summary/Conclusions

This Bachelor-thesis was about increasing security in PeerTrust, a trust negotiation software written in Java that allows an automatic exchange of rules, policies and credentials between parties in order to gain access to a sensitive resource. This is an alternative to traditional client/server-systems where the user has to manually register and login.

The first task was the implementation of the *authenticatesTo(Identity,Party,Authority)@Requester*-predicate which is evaluated to true if *Requester* can show a X.509 certificate chain (that states that he has identity *Identity* issued by authority *Authority*, so the first certificate's issuer must be *Identity* and the last one's issuer *Authority*) to *Party*. The first approach uses the TLS-handshake, because the transmission of specified certificate chains as well the proof from *Requester* that he really owns the private key that corresponds public key of the first certificate of the chain (an important concept) are already integrated in it. As this approach can't be easily integrated in the existing communication implementation and needs an additional serversocket, the predicate is reimplemented without the using the TLS-handshake, this approach hasn't the disadvantages just mentioned. In it, the construction and transmission of the suitable certificate chain as well the private-key-proof are reimplemented manually.

When receiving an answer, a party may want to verify it because there is no guarantee that other parties don’t lie. Each answer has a proof tree (contains all necessary rules used, credentials, certificates or certificate chains) which can be used for that. The second task was to implement this verification of a proof tree, so that parties can decide if the answer is correct or not. All rules of the proof tree are added to the knowledge base of an inference engine and it checks if the last expanded goal can be inferenced from them by using a special predicate. If a rule has credentials or certificate chains (signed rules or the authenticatesTo-predicate), these are also checked.

The third task was to verify credentials. For PeerTrust, credentials are texts in the extensions of X.509 certificates that belong to signed rules. So not only the text has to be checked, if it corresponds to the signed rule, also the certificate itself is investigated if it isn't expired and can be verified with the suitable public key. This one is contained in a certificate chain, which must also be checked. The solution of this task is also used in the second one (verification of the proof tree).

All these three concepts increase security and diminish misuse. A party may have more than one private key and corresponding certificate (chain), so the authenticatesTo-predicate can be used more than once to force him to show a different certificate chain (issued by a different authority) each time. Additional authentications make it harder for an attacker to manipulate a trust negotiation, because he will need almost all (not just one) certificates and private keys of a party to

fake it's identity. When an answer is faked or not correct, this can be detected by the receiver if he verifies the associated proof tree and the credentials included in it.

The implementation of the authenticatesTo-predicate and the verification of credentials only work with X.509 credentials. Interesting topics for future work would be the support of other types of authentication, credentials or certificates, maybe the authenticatesTo-predicate can have an additional parameter that specifies which authentication method or certificate type must be used. This would lead to more flexibility and security.

7. References

1. J. Basney, W. Nejdl, D. Olmedilla, V. Welch, and M. Winslett.
Negotiating Trust on the Grid.
In Proc. of 2nd Workshop on Semantics in P2P and Grid Computing at the Thirteenth International World Wide Web Conference, May 2004, New York, USA
http://www.learninglab.de/~olmedilla/pub/negotiationOnTheGrid.pdf

2. R. Gavriloaie, W. Nejdl, D. Olmedilla, K. Seamons, and M. Winslett.
No Registration Needed: How to Use Declarative Policies and Negotiation to Access Sensitive Resources on the Semantic Web.
In Proc. of 1st European Semantic Web Symposium, May. 2004, Heraklion, Greece
http://www.l3s.de/~olmedilla/pub/PeerTrust-NoRegistration.pdf

3. W. Nejdl, D. Olmedilla, and M. Winslett.
PeerTrust: Automated Trust Negotiation for Peers on the Semantic Web.
Technical Report
http://www.l3s.de/~olmedilla/pub/PeerTrust-ATN.pdf

4. *J2SE 1.4.2 API Specification.*
http://java.sun.com/j2se/1.4.2/docs/api/

5. *Java Secure Socket Extension (JSSE) Reference Guide for the Java 2 SDK, Standard Edition, v 1.4.2.*
http://java.sun.com/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html

6. A. Hess, Jared Jacobson, H. Mills, R. Wamsley, K. Seamons, and B. Smith.
Advanced Client/Server Authentication in TLS.
In 8th ACM Symposium on Access Control Models and Technologies, Como, Italy, June 2003.
http://www.isoc.org/isoc/conferences/ndss/02/proceedings/papers/hess.pdf

7. M. Winslett.
An Introduction to Trust Negotiation.
In Workshop on Credential-Based Access Control, Dortmund, October 2002
http://dais.cs.uiuc.edu/pubs/winslett/itrust03.ps

8. J. Trevor and D. Suciu.
Dynamically distributed query evaluation.
In Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Santa Barbara, CA, USA, May 2001.
http://www.research.att.com/~trevor/papers/JimPODS2001.pdf

9. M. Y. Becker and P. Sewell.
Cassandra: distributed access control policies with tunable expressiveness.
In Policies in Distributed Systems and Networks, June 2004.
http://www.cl.cam.ac.uk/users/mywyb2/publications/becker04cassandra-csfw2004.pdf

10. T. Yu, M.Winslett and K. Seamons.
Supporting Structured Credentials and Sensitive Policies through Interoperable Strategies in Trust Negotiation
In ACM Transactions on Information and System Security, 6(1), Feb. 2003.
http://www4.ncsu.edu:8030/~tyu/pubs/tissec03.pdf

11. N. Li, J. Mitchell and W. Winsborough.
Design of a Role-based Trust-management Framework

In IEEE Symposium on Security and Privacy, Berkeley, California, May 2002.
http://crypto.stanford.edu/~ninghui/papers/rt_oakland02.pdf

12. N. Li, W. Winsborough and J.Mitchell.
Distributed Credential Chain Discovery in Trust-Management
In Journal of Computer Security, 11(1), Feb. 2003.
http://crypto.stanford.edu/~ninghui/papers/discovery_jcs03.pdf

13. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*
http://www.ietf.org/rfc/rfc2459.txt

14. *The TLS Protocol Version 1.0*
http://www.ietf.org/rfc/rfc2246.txt

15. *PureTLS*
http://www.rtfm.com/puretls/

16. T. Jim.
SD3: A Trust Management System With Certified Evaluation.
In IEEE Symposium on Security and Privacy, Oakland, CA, May 2001.
http://www.research.att.com/~trevor/papers/JimOakland2001.pdf

8. Appendix A

This chapter contains some important classes that were newly created during the work on this Bachelor-thesis. The websites that the javadoc-tool generates are presented here, these can also be found on the included CD (as an extended version with all classes that were manipulated during the bachelor-thesis because these can't be shown all in this appendix) which also contains the complete source code, test classes etc.

org.peertrust.security

Class AuthenticatesTo

```
java.lang.Object
  |
  +--org.peertrust.security.AuthenticatesTo
```

public class **AuthenticatesTo**
extends java.lang.Object

This class contains the complete handling of the authenticatesTo-predicate. Its methods must be called from the MetaInterpreter- and MetaInterpreterListener-classes. Exact syntax for predicate: authenticatesTo(Identity,Party,Authority)@Requester

Field Summary

static java.lang.String	**AUTH_PREDICATE** Name of the predicate that is used between Requester and Party in the authenticatesTo-predicate-handling in order to communicate.
static java.lang.String	**AUTH_TO_PREDICATE** Name of the authenticatesTo-predicate.
static java.lang.String	**CRYPT_ALGORITHM** Name of the algorithm that should be used for encryption.

Constructor Summary

AuthenticatesTo()	

Method Summary

static boolean	**authenticatesPredicate**(org.peertrust.meta.Tree tree, net.jxta.edutella.util.Configurator config, org.peertrust.net.Peer localPeer, org.peertrust.meta.MetaInterpreter metainterpreter) If the authentication-message is contained in the Tree-object, the local party must authenticate himself right in order to satisfy the authenticatesTo-predicate.
static boolean	**authenticatesToPredicate**(org.peertrust.meta.Tree tree, org.peertrust.meta.MetaInterpreter metainterpreter, java.util.Hashtable entities, org.peertrust.net.Peer localPeer, net.jxta.edutella.util.Configurator config, org.peertrust.strategy.Queue queue) Looks if an authenticatesTo-predicate appears in the Tree's subgoals.
static boolean	**authentication_answerPredicate**(org.peertrust.meta.Tree tree, org.peertrust.net.Answer answer, org.peertrust.inference.InferenceEngine engine) Checks, if the current answer contains an authentication-predicate and satisfies an authenticatesTo-predicate that must be evaluated by the local peer.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

CRYPT_ALGORITHM

public static final java.lang.String **CRYPT_ALGORITHM**

Name of the algorithm that should be used for encryption.

See Also:
Constant Field Values

AUTH_TO_PREDICATE

public static final java.lang.String **AUTH_TO_PREDICATE**

Name of the authenticatesTo-predicate.

See Also:
Constant Field Values

AUTH_PREDICATE

```
public static final java.lang.String AUTH_PREDICATE
```

Name of the predicate that is used between Requester and Party in the authenticatesTo-predicate-handling in order to communicate.

See Also:
Constant Field Values

Constructor Detail

AuthenticatesTo

```
public AuthenticatesTo()
```

Method Detail

authenticatesToPredicate

```
public static boolean authenticatesToPredicate(org.peertrust.meta.Treetree,
                                               org.peertrust.meta.MetaInterprete
rmetainterpreter,
                                               java.util.Hashtableentities,
                                               org.peertrust.net.PeerlocalPeer,
                                               net.jxta.edutella.util.Configurat
orconfig,
                                               org.peertrust.strategy.Queuequeue
)
```

Looks if an authenticatesTo-predicate appears in the Tree's subgoals. If yes, this predicate is either delegated to the right party or Party is informed how he must authenticate himself in order to satisfy the predicate. This method must be called from the MetaInterpreter.

Parameters:
`tree` - The Tree-object that is should be examined.
`metainterpreter` - The local MetaInterpreter-object.
`localPeer` - The local peer.
`config` - The config-object.
`queue` - The queue the MetaInterpreter uses.

Returns:
boolean If an authenticatesTo-predicate was contained in the Tree-object's subgoals or not.

---

authenticatesPredicate

```
public static boolean authenticatesPredicate(org.peertrust.meta.Treetree,
                                             net.jxta.edutella.util.Configurator
```

```
config,
                                                    org.peertrust.net.PeerlocalPeer,
                                                    org.peertrust.meta.MetaInterpreterm
etainterpreter)
```

If the authentication-message is contained in the Tree-object, the local party must authenticate himself right in order to satisfy the authenticatesTo-predicate. This method must be called from the MetaInterpreter.

Parameters:

`tree` - The Tree-object that is should be examined.
`config` - The config-object.
`localPeer` - The local peer.
`metainterpreter` - The local MetaInterpreter-object.

Returns:

boolean If an authentication-message was contained in the Tree-object's subgoals or not.

---

authentication_answerPredicate

```
public static boolean authentication_answerPredicate
(org.peertrust.meta.Treetree,
                                                    org.peertrust.net.Answerans
wer,
                                                    org.peertrust.inference.Inf
erenceEngineengine)
```

Checks, if the current answer contains an authentication-predicate and satisfies an authenticatesTo-predicate that must be evaluated by the local peer. This method must be called from the MetaInterpreterListener.

Parameters:

`tree` - The Tree-object that matches to the answer.
`answer` - The current answer.
`engine` - The inference engine.

Returns:

boolean Is false, if this is an authentication-answer, but doesn't satisfy the authenticatesTo-predicate, otherwise true.

---

org.peertrust.security

Class CertificateChain

```
java.lang.Object
  |
  +--org.peertrust.security.CertificateChain
```

All Implemented Interfaces:
java.io.Serializable

---

public class **CertificateChain**
extends java.lang.Object
implements java.io.Serializable

This class represents a X.509 certificate chain. It offers two different approaches for automatically building such a chain out of the entries in the keystore.

See Also:
Serialized Form

Field Summary

private java.security.cert.X509Certificate[]	**x509certificate** The certificate chain.

Constructor Summary

CertificateChain(net.jxta.edutella.util.Configurator config, java.lang.String identity, java.lang.String firstissuer, java.lang.String authority) Constructor.

Method Summary

java.security.cert.X509Certificate[]	**getCertificates**() Returns the certificate chain as an array.
static java.lang.String	**getIssuerAlias**(java.security.cert.X509Certificate cert) Returns the issuer-alias of a certificate.
static java.lang.String	**getSubjectAlias**(java.security.cert.X509Certificate cert) Returns the subject-alias of a certificate.
private boolean	**isCertificateValid**(java.security.cert.X509Certificate cert) Checks if a given certificate is still valid or expired.

private void	**searchComplexChain**(java.lang.String identity, java.lang.String authority, java.security.KeyStore ks, java.util.Vector vectorCerts) A complex concept for building certificate chains.
private void	**searchSimpleChain**(java.lang.String identity, java.lang.String firstissuer, java.security.KeyStore ks, java.util.Vector vectorCerts) A simple concept for building certificate chains.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

x509certificate

```
private java.security.cert.X509Certificate[] x509certificate
```

The certificate chain.

Constructor Detail

CertificateChain

```
public CertificateChain(net.jxta.edutella.util.Configuratorconfig,
                        java.lang.Stringidentity,
                        java.lang.Stringfirstissuer,
                        java.lang.Stringauthority)
```

Constructor. Builds and stores the certificate chain.

Parameters:

`config` - The configuration-object of the local peer.
`identity` - The subject that the first certificate must have, null, if it doesn't matter.
`firstissuer` - The issuer that the first certificate must have, null, if it doesn't matter.
`authority` - The issuer that the last certificate of the chain must have, null, if it doesn't matter.

Method Detail

searchSimpleChain

```
private void searchSimpleChain(java.lang.Stringidentity,
```

```
                                java.lang.Stringfirstissuer,
                                java.security.KeyStoreks,
                                java.util.VectorvectorCerts)
                         throws java.lang.Exception
```

A simple concept for building certificate chains. If an authority wasn't specified in the constructor, this method is called.

Parameters:

`identity` - The subject that the first certificate must have, null, if it doesn't matter.
`firstissuer` - The issuer that the first certificate must have, null, if it doesn't matter.
`vectorCerts` - An out-parameter that contains the resulting chain after this method ends.
`java.lang.Exception`

---

searchComplexChain

```
private void searchComplexChain(java.lang.Stringidentity,
                                java.lang.Stringauthority,
                                java.security.KeyStoreks,
                                java.util.VectorvectorCerts)
                         throws java.lang.Exception
```

A complex concept for building certificate chains. If an authority is specified in the constructor, this method is called.

Parameters:

`identity` - The subject that the first certificate must have, null, if it doesn't matter.
`authority` - The issuer that the last certificate of the chain must have.
`vectorCerts` - An out-parameter that contains the resulting chain after this method ends.
`java.lang.Exception`

---

getCertificates

```
public java.security.cert.X509Certificate[] getCertificates()
```

Returns the certificate chain as an array.

Returns:

X509Certificate[] Chain as an Array of certificates.

---

isCertificateValid

```
private boolean isCertificateValid(java.security.cert.X509Certificatecert)
```

Checks if a given certificate is still valid or expired.

Parameters:
`cert` - The certificate that should be checked.
Returns:
boolean true, if certificate is still valid, false, if it's expired.

---

getSubjectAlias

```
public static java.lang.String getSubjectAlias
(java.security.cert.X509Certificatecert)
```

Returns the subject-alias of a certificate.

Parameters:
`cert` - The certificate which alias should be returned.
Returns:
String the subject-alias of the certificate.

---

getIssuerAlias

```
public static java.lang.String getIssuerAlias
(java.security.cert.X509Certificatecert)
```

Returns the issuer-alias of a certificate.

Parameters:
`cert` - The certificate which alias should be returned.
Returns:
String the issuer-alias of the certificate.

Overview Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.peertrust.security

Class ProofTreeValidator

```
java.lang.Object
  |
  +--org.peertrust.security.ProofTreeValidator
```

---

public final class **ProofTreeValidator**
extends java.lang.Object

This class is responsible for verifying the proof tree of a query or answer. A party can detect so, if the answer is really valid.

Constructor Summary

ProofTreeValidator()	

Method Summary

private static boolean	**checkAuthenticatesToProofRule** (AuthenticatesToProofRule authrule, net.jxta.edutella.util.Configurator config) Checks if the answer to a authenticatesTo-predicate and the corresponding certificate chain are correct in the proof tree.
private static boolean	**checkLastCertificate** (java.security.cert.X509Certificate cert, net.jxta.edutella.util.Configurator config) Checks if the suitable public key for the issuer of a given certificate can be found in the keystore and the certificate can be verified with it.
private static boolean	**checkSignedProofRule**(SignedProofRule signedproofrule) Checks if a signed rule and the corresponding credential are correct in the proof tree.
static boolean	**isProofTreeOk**(org.peertrust.meta.Tree tree, net.jxta.edutella.util.Configurator config) Checks, if a Tree is correct by verifying its proof tree.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

ProofTreeValidator

```
public ProofTreeValidator()
```

Method Detail

isProofTreeOk

```
public static boolean isProofTreeOk(org.peertrust.meta.Treetree,
                                    net.jxta.edutella.util.Configuratorconfig)
```

Checks, if a Tree is correct by verifying its proof tree.

Parameters:
`tree` - The tree whose proof tree should be verified.
`config` - The configuration-object of the local peer.
Returns:
boolean If the Tree has a valid proof tree.

---

checkSignedProofRule

```
private static boolean checkSignedProofRule(SignedProofRulesignedproofrule)
```

Checks if a signed rule and the corresponding credential are correct in the proof tree.

Parameters:
`signedproofrule` - The signed rule that should be checked.
Returns:
boolean If the signed rule in the proof tree is correct or not.

---

checkAuthenticatesToProofRule

```
private static boolean checkAuthenticatesToProofRule
(AuthenticatesToProofRuleauthrule,
                                                    net.jxta.edutella.util.Conf
iguratorconfig)
```

Checks if the answer to a authenticatesTo-predicate and the corresponding certificate chain are correct in the proof tree.

Parameters:
`authrule` - The answer to the authenticatesTo-predicate that should be checked.
`config` - The configuration-object of the local peer.
Returns:
boolean If the answer to the predicate is correct or not.

checkLastCertificate

```
private static boolean checkLastCertificate
(java.security.cert.X509Certificatecert,
                                                net.jxta.edutella.util.Configuratorc
onfig)
```

Checks if the suitable public key for the issuer of a given certificate can be found in the keystore and the certificate can be verified with it.

Parameters:

`cert` - The certificate that should be verified.

`config` - The configuration-object of the local peer.

Returns:

boolean If the public key can be found and the certificate can be verfied with it.

Overview Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.peertrust.security

Class ProofRule

```
java.lang.Object
  |
  +--org.peertrust.security.ProofRule
```

All Implemented Interfaces:

java.io.Serializable

Direct Known Subclasses:

AuthenticatesToProofRule, SignedProofRule

---

public class **ProofRule**
extends java.lang.Object
implements java.io.Serializable

Represents a local rule (not a signed or authenticatesTo one) in a proof tree.

See Also:
Serialized Form

Field Summary

private java.lang.String	**strRule** The rule as text representation.

Constructor Summary

ProofRule(java.lang.String rule) Constructor.	

Method Summary

java.lang.String	**getRule**() Returns the rule as text.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

strRule

```
private java.lang.String strRule
```

The rule as text representation.

Constructor Detail

ProofRule

```
public ProofRule(java.lang.Stringrule)
```

Constructor.

Parameters:
`rule` - The local rule as text representation.

Method Detail

getRule

```
public java.lang.String getRule()
```

Returns the rule as text.

Returns:
String The rule as text.

Overview Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.peertrust.security

Class SignedProofRule

```
java.lang.Object
  |
  +--org.peertrust.security.ProofRule
        |
        +--org.peertrust.security.SignedProofRule
```

All Implemented Interfaces:
java.io.Serializable

---

public class **SignedProofRule**
extends ProofRule
implements java.io.Serializable

Represents a signed rule in a proof tree. Contains also the corresponding credential and certificate chain.

See Also:
Serialized Form

Field Summary

private CertificateChain	**certificatechain** The CertificateChain that corresponds to the signed rule.
private org.peertrust.security.credentials.Credential	**credential** The credential that corresponds to the signed rule.

Fields inherited from class org.peertrust.security.ProofRule

Constructor Summary

SignedProofRule(java.lang.String rule, org.peertrust.security.credentials.Credential credential, net.jxta.edutella.util.Configurator config) Constructor.

Method Summary

CertificateChain	**getCertificateChain**() Return the corresponding certificate chain.
org.peertrust.security.credentials.Credential	**getCredential**() Return the corresponding credential.

Methods inherited from class org.peertrust.security.ProofRule
getRule

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

credential

```
private org.peertrust.security.credentials.Credential credential
```

The credential that corresponds to the signed rule.

---

certificatechain

```
private CertificateChain certificatechain
```

The CertificateChain that corresponds to the signed rule.

Constructor Detail

SignedProofRule

```
public SignedProofRule(java.lang.Stringrule,
                       org.peertrust.security.credentials.Credentialcredential,
                       net.jxta.edutella.util.Configuratorconfig)
```

Constructor. Constructs the certificate chain.

Parameters:

`rule` - The signed rule in text representation.
`credential` - The corresponding credential.

Method Detail

getCredential

```
public org.peertrust.security.credentials.Credential getCredential()
```

Return the corresponding credential.

Returns:

Credential The corresponding credential.

---

getCertificateChain

```
public CertificateChain getCertificateChain()
```

Return the corresponding certificate chain.

Returns:

CertificateChain The corresponding certificate chain.

---

org.peertrust.security

Class AuthenticatesToProofRule

```
java.lang.Object
  |
  +--org.peertrust.security.ProofRule
        |
        +--org.peertrust.security.AuthenticatesToProofRule
```

All Implemented Interfaces:
java.io.Serializable

public class **AuthenticatesToProofRule**
extends ProofRule

This class represents the proof of an authenticatesTo-predicate in a proof tree. It contains the certificate chain that should correspond to the parameters of this predicate.

See Also:
Serialized Form

Field Summary

private CertificateChain	**certificatechain** The certificate chain that should satisfy the authenticatesTo-predicate.

Fields inherited from class org.peertrust.security.ProofRule

Constructor Summary

AuthenticatesToProofRule(java.lang.String rule, CertificateChain chain) Constructor.

Method Summary

CertificateChain	**getCertificateChain**() Returns the certificate chain.

Methods inherited from class org.peertrust.security.ProofRule
getRule

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

certificatechain

```
private CertificateChain certificatechain
```

The certificate chain that should satisfy the authenticatesTo-predicate.

Constructor Detail

AuthenticatesToProofRule

```
public AuthenticatesToProofRule(java.lang.Stringrule,
                                CertificateChainchain)
```

Constructor.

Parameters:

`rule` - The authenticatesTo-predicate as text.
`chain` - The certificate chain that should satisfy the predicate.

Method Detail

getCertificateChain

```
public CertificateChain getCertificateChain()
```

Returns the certificate chain.

Returns:

CertificateChain The certificate chain that should satisfy the predicate.
